C++ Lambda Story

Everything you need to know about **Lambda Expressions** in **Modern C++**!

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

Bartłomiej Filipek

(BF) C++ STORIES

C++ Lambda Story

Everything you need to know about Lambda Expression in Modern C++!

Bartłomiej Filipek

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

This version was published on 2020-06-19

Leanpub

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

About the Book . . . . **i**

Roots Of The Book i

Who This Book is For ii

Reader Feedback ii

About the Author . . . . **v**

Acknowledgement . . . . **vi**

Revision History . . . . **vii**

1. Lambdas in C++03 . . . . **1**

Callable Objects in C++03 2

Issues with Functors 5

Composing With Functional Helpers 6

Motivation for a New Feature 8

2. Lambdas in C++11 . . . . **10**

The Syntax 11

The Type of a Lambda 13

The Call Operator 15

Captures 15

Return Type 25

Conversion to a Function Pointer 28

IIFE - Immediately Invoked Function Expression 29

Inheriting from a Lambda 31

Summary 35

3. Lambdas in C++14 . . . . **36**

Default Parameters for Lambdas 37

Return Type 37
Captures With an Initialiser 38
Capturing a Member Variable 41
Generic Lambdas 42
Replacing `std::bind1st` and `std::bind2nd` with Lambdas 44
LIFTing with lambdas 47
Summary 48

4. Lambdas in C++17 . . . . **49**
`constexpr` Lambda Expressions 50
Capture of `*this` 52
Updates To IIFE 54
Deriving from Multiple Lambdas 55
Summary 61

5. Lambdas in C++20 . . . . **62**
A Quick Overview of the Changes 63
Template Lambdas 65
Concepts and Lambdas 68
Changes to Stateless Lambdas 70
Lambdas and `constexpr` Algorithms 73
C++20 Updates to the Overloaded Pattern 74
Summary 75

Appendix A - List of Techniques **76**

Appendix B - Top Five Advantages of C++ Lambda Expressions **77**
1. Lambdas Make Code More Readable 77
2. Lambdas Improve Locality of the Code 79
3. Lambdas Allow to Store State Easily 79
4. Lambdas Allow Several Overloads in the Same Place 80
5. Lambdas Get Better with Each Revision of C++! 82
Your Turn 83

References **84**

About the Book

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

- C++03 - how to code without lambda support? What was the motivation for the new modern C++ feature?
- 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.

Additionally, throughout the chapters, you'll find techniques and handy patterns for using lambda in your code.

Roots Of The Book

The idea for the content started after a live coding presentation given by Tomasz Kamiński at our local Cracow C++ User Group.

I took the ideas from the presentation and then created 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]

Then, I decided that I want to offer my readers not only blog posts but a nice-looking PDF. Leanpub provides an easy way to create such PDFs, so it was the right choice to copy the articles' content and create a Leanpub book.

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

Why not move further?

After some time, I decided to write more content, update the examples, provide better use cases and patterns. And here you have the book! It's now even twice the size of the initial material that is available on the blog!

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.

Reader Feedback

If you spot an error, a typo, a grammar mistake… or anything else (especially logical issues!) that should be corrected, then please send your feedback to bartlomiej.filipek AT bfilipek.com.

You can also use this place:

- Leanpub Book's Feedback Page[3]

What's more, the book has a dedicated page at GoodReads. Please share your review there:

- C++ Lambda Story @GoodReads[4]

Code License

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

Formatting

The code is presented in a monospaced font, similar to the following example:

For longer examples:

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

[4]https://www.goodreads.com/book/show/53609731-c-lambda-story

Title Of the Example

```
#include <iostream>

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

Or shorter snippets:

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

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)[5]
- The first method of a class is not highlighted - #1084 - First method of class not highlighted in C++[6]
- Teplate method is not highlighted #1434 - C++ lexer doesn't recognize function if return type is templated[7]
- Modern C++ attributes are sometimes not recognised properly

Other issues for C++ and Pygments: issues C++[8].

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

Online Compilers

Instead of creating local projects to play with the code samples, you can also leverage 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 convenient if you want to play with code samples and check the results using various compilers.

For example, many of the code samples for this book were created using Coliru Online and Wandbox compilers and then adapted for the book.

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

- Coliru[9] - uses GCC 8.2.0 (as of July 2019), offers link sharing and a basic text editor, it's simple but very effective.
- Wandbox[10] - offers a lot of compilers, including most Clang and GCC versions, can use boost libraries; offers link sharing and multiple file compilation.
- Compiler Explorer[11] - offers many compilers, shows compiler output, can execute the code.
- CppBench[12] - runs simple C++ performance tests (using google benchmark library).
- C++ Insights[13] - a Clang-based tool for 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 helpful list of online compilers gathered on this website: List of Online C++ Compilers[14].

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

About the Author

Bartłomiej (Bartek) Filipek is a C++ software developer with more than 12 years of professional experience. In 2010 he graduated from Jagiellonian University in Cracow, Poland 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 bfilipek.com. Initially, the topics revolved around graphics programming, but now the blog focuses on core C++. He's also a co-organiser of the C++ User Group in Cracow. 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 the Polish National Body which works directly with ISO/IEC JTC 1/SC 22 (C++ Standardisation Committee). In the same month, Bartek was awarded his first MVP title for the years 2019/2020 by Microsoft.

In his spare time, he loves collecting and assembling Lego models with his little son.

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!
- 5th January 2020 - Grammar, style, example updates, wording, IIFE sections, C++20 updates
- 17th April 2020 - Improved and extended descriptions of new features in the C++20 Chapter, grammar, wording, layout
- 30th April 2020 - Added several sections about Deriving from lambda expressions, in C++11, C++17 and C++20 chapters.
- 19th June 2020 - Major update
  - Improved C++03 chapter, added sections about helper functional objects from the Standard Library,
  - Added new section on how to convert from deprecated `bind1st` into modern alternatives in the C++14 chapter
  - improved and extended IFFE section in C++11 and C++17 chapters
  - new Appendix with a list of used techniques
  - new Appendix with a list of top 5 lambda features, adapted from a blog article
  - new title image with updated subtitle
  - lots of smaller improvements across the whole book

1. Lambdas in C++03

At first, it's good to create some background for our main topic. To do this, we'll move into the past and look at the code that doesn't use any modern C++ techniques.

In this chapter, you'll learn:

- How to pass functors to algorithms from the Standard Library
- What are the limitations of functors and function pointers
- Why functional helpers weren't good enough
- What was the motivation for the new feature for C++0x/C++11

Callable Objects in C++03

One of the fundamental ideas of the Standard Library is that algorithms like `std::sort`, `std::for_each`, `std::transform` and many others, can take any callable object and call it on elements of the input container. However, in C++03, this only included pointers to functions and functors.

As an example, let's have a look at an application that prints all elements of a vector.

In the first version we'll use a regular function:

A basic print function

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

void PrintFunc(int x) {
    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(), PrintFunc);
}
```

Runnable code: @Wandbox[1]

The code above uses `std::for_each` to iterate over a vector (we use the C++03 version so range-based for loop is not available!) and then it passes `PrintFunc` as a callable object.

We can convert this simple function into a functor:

[1] https://wandbox.org/permlink/XiMBBTOG122vplUS

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[2]

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:

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;
    }
```

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

```
    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[3]

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

As you can easily predict, we should get the following output:

```
1
2
num calls: 2
```

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.

Functor with a 'captured' variable

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

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

    void operator()(int x) const {
```

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

```
        std::cout << strText << x << '\n';
        ++numCalls;
    }

    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[4]

In this version, `PrintFunctor` takes an extra parameter to initialise a member variable. Then this variable is used in the call operator. So the expected output is as follows:

```
Elem: 1
Elem: 2
num calls: 2
```

Issues with Functors

As you can see, functors are powerful. They are represented by a separate class, and you can design them any way you like.

However, in C++03, the problem was that you had to write a function or a functor in a different place than the invocation of the algorithm. This could mean that the code for a function could be dozens or hundreds of lines later earlier or further in the source file.

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

As a potential solution, you might have tried 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.

But C++ programmers are smart and found some ways to work around the issues with C++03. See the next section below.

Composing With Functional Helpers

How about having some helpers and predefined functors?

If you check the `<functional>` header from the Standard Library, you'll find a lot of types and functions that can be immediately used with the standard algorithms.

For example:

- `std::plus<T>()` - takes two arguments and returns their sum.

- `std::minus<T>()` - takes two arguments and returns their difference.
- `std::less<T>()` - takes two arguments and returns if the first one is smaller than the second
- `std::greater_equal<T>()` - takes two arguments and returns if the first is greater or equal to the second.
- `std::bind1st` - creates a callable object with the first argument fixed to the given value.
- `std::bind2nd` - creates a callable object with the second argument fixed to the given value.
- and many more

Let's write some code that benefits from the helpers:

Using old C++03 functional helpers

```
#include <algorithm>
#include <functional>
#include <vector>

int main() {
    std::vector<int> v { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    const auto smaller5 = std::count_if(v.begin(), v.end(),
                                        std::bind2nd(std::less<int>(), 5));

    return smaller5;
}
```

The example uses `std::less` and then fixes its second argument by using `std::bind2nd` [5]. As you can probably guess, the code expands into a function that performs a simple comparison:

```
return x < 5;
```

If you wanted more ready-to-use helpers, then you can also look at the boost library, for example `boost::bind`.

[5] `bind1st`, `bind2nd` and other functional helpers were deprecated in C++11 and removed in C++17. The code in this chapter uses them only to illustrate C++03 issues. Please use some modern alternatives in your projects. See the C++14 chapter for more information.

Unfortunately, the main issue with this approach is the complexity and hard-to-learn syntax. For instance, writing code that composes two or more functions is not natural. Have a look below:

Composing functional helpers

```
using std::placeholders::_1;

const std::vector<int> v { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
const auto val = std::count_if(v.begin(), v.end(),
                               std::bind(std::logical_and<bool>(),
                               std::bind(std::greater<int>(),_1, 2),
                               std::bind(std::less_equal<int>(),_1,6)));

// _1 comes from the std::placeholder namespace
```

Play with the code @Compiler Explorer[6]

The composition uses `boost::bind` with `std::greater` and `std::less_equal` connected with `std::logical_and`. Additionally, the code uses `_1` which is a placeholder for the first input argument.

While the above code works, and you can define it locally, you probably agree that it's complicated and not natural syntax. Not to mention that this composition represents only a simple condition:

```
return x > 2 && x <= 6;
```

Is there anything better and more natural to use?

Motivation for a New Feature

As you can see, in C++03, there were several ways to declare and pass a callable object to algorithms and utilities from the Standard Library. However, all of those options were a bit limited. For example, you couldn't declare a local functor object, or it was complicated to compose a function with functional helper objects.

Fortunately with C++11 we finally saw a lot of improvements!

[6] https://godbolt.org/z/_9Ptzg

First of all, the C++ Committee lifted the limitation of the template instantiation with a local type. Now you can write functors locally, in the place where you need them.

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

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

If we look at N3337[7] - the final draft of C++11, we can see a separate section for lambdas: [expr.prim.lambda][8].

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

[7]https://timsong-cpp.github.io/cppwp/n3337/
[8]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++.

You 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 incorporate new syntax, but then the compiler "expands" it into an unnamed "hidden" functor object. This way we have all advantages (and disadvantages) 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
- How lambda can be converted to a function pointer and use it with C-style API
- What's IIFE
- How to inherit from a lambda and why it can be useful

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 shows how to write a lambda expression and pass it to `std::for_each`. For comparison, the code also illustrates the corresponding functor type:

First Lambda and a Corresponding Functor

```
#include <algorithm>
#include <iostream>
#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 specification, trailing return type, ...
|  |
|  optional: parameter list
|
lambda introducer with a 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**.

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

A few examples of lambda expressions:

```
// 1. the simplest lambda:
[]{}
```

In the first example, you can see a "minimal" lambda expression. It only needs the `[]` and then the empty `{}` for the function body.

```
// 2. with two params:
[](float f, int a) { return a*f; }
[](int a, int b) { return a < b; }
```

In the second example, probably one of the most common, you can see that the arguments are passed into the `()` section, just like for a regular function. The return type is not needed, as the compiler will automatically deduce it.

```
// 3. trailing return type:
[](MyClass t) -> int { auto a = t.compute(); print(a); return a; }
```

In the above example, we explicitly set a return type. The trailing return type is also available for regular function declaration since C++11.

```
// 4. additional specifiers:
[x](int a, int b) mutable { ++x; return a < b; }
[](float param) noexcept { return param*param; }
```

The last example shows that before the body of the lambda, you can use other specifiers. In the code, we used `mutable` (so that we can change the captured variable) and also `noexcept`.

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.

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

We can also read in: [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:

```cpp
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:

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

Their types are different! Even if the "code behind" is the same, the compiler is required to declare two unique unnamed types for each lambda.

You can, however copy lambdas:

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

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 str {"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-place" of lambda declaration.

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

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

> When the lambda-expression is evaluated, the entities that are captured by copy are used to direct-initialise 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;
};
```

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

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.

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

While specifying `[=]` or `[&]` might be convenient, as it captures all automatic storage duration variables, 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.

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.

Invocation Counter - An Example of Captured Variables

Before we move on to some more complicated topics with capturing, we can have a little break and focus on a more practical example.

Lambda expressions are handy when you want to use some existing algorithm from the Standard Library and alter the default behaviour. For example, for `std::sort` you can write your comparison function.

But we can go further and enhance the comparator with an invocation counter. Have a look:

Invocation Counter

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

int main() {
    std::vector<int> vec { 0, 5, 2, 9, 7, 6, 1, 3, 4, 8 };

    size_t compCounter = 0;
    std::sort(vec.begin(), vec.end(), [&compCounter](int a, int b) {
        ++compCounter;
        return a < b;
    });

    std::cout << "number of comparisons: " << compCounter << '\n';

    for (auto& v : vec)
        std::cout << v << ", ";
}
```

You can play with the code @Compiler Explorer[9]

The comparator provided in the example works in the same way as the default one, it returns if `a` is smaller than `b`, so we use the natural order from lowest to the largest numbers. However, the lambda passed to `std::sort` also captures a local variable `compCounter`. The variable is then used to count all of the invocations of this comparator from the sorting algorithm.

Capturing Global Variables

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[10]

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://godbolt.org/z/Ashkgp
[10] https://wandbox.org/permlink/hsS8K0I6PrRyX45Z

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

See @Wandbox[11]

Capturing Statics

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

Capturing Static Variables

```cpp
#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[12]

The output:

```
10
11
12
```

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

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

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[13]

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

```
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:

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

```
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 <functional>
#include <iostream>

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[14]

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

```
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.

Moveable-only Objects

If you have an object that is moveable only (for example `unique_ptr`), then you cannot move it to lambda as a captured variable. Capturing by value does not work; you can only capture by reference.

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

```
std::unique_ptr<int> p(new int{10});
auto foo = [p]() {}; // does not compile....
auto foo_ref = [&p]() { }; // compiles, but the ownership
                           // is not passed
```

In the above example, you can see that the only way to capture `unique_ptr` is by reference. This approach, however, might not be the best as it doesn't transfer the ownership of the pointer.

In the next chapter, about C++14, you'll see that this issue is fixed thanks to the capture with initialiser.

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 << '\n';
    // x = 11; // won't compile as it's const!
};
foo();
```

Test code @Wandbox[15]

Return Type

In many cases, you can skip the return type of the lambda and then the compiler will deduce the type for you.

Initially, return type deduction was restricted to lambdas with bodies containing a single return statement. However, 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[16] [17]

To sum up, since C++11, the compiler can deduce the return type as long as all of your return statements are of the same type.

[15] https://wandbox.org/permlink/pbnGo223HNdOoNLQ
[16] http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_defects.html#975
[17] Thanks to Tomek for finding the correct link!

> 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[18]

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 adapt to the rules of `auto` type deduction for regular functions.

Trailing Return Type Syntax

If you want to be explicit about the return type, you can use trailing return type specification. For example, when you return a string literal:

[18] https://wandbox.org/permlink/kVKjlBObC9futJNV

Returning a string literal from a lambda

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

int main() {
    auto testSpeedString = [](int speed) {
        if (speed > 100)
            return "you're a super fast";

        return "you're a regular";
    };

    auto str = testSpeedString(100);
    str += " driver";                    // uups! no += on const char*!

    std::cout << str;

    return 0;
}
```

The above code doesn't compile because the compiler deduces `const char*` as the return type for the lambda. Since there's no `+=` operator available on string literals, then the code breaks.

We can fix the problem by explicitly setting the return type:

```
auto testSpeedString = [](int speed) -> std::string {
    if (speed > 100)
        return "you're a super fast";

    return "you're a regular";
};

auto str = testSpeedString(100);
str += " driver"; // works fine
```

You can play with the code @Coliru[19]

[19] http://coliru.stacked-crooked.com/a/45cebc8b35d5b2a9

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

```
#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[20]

[20] https://wandbox.org/permlink/XAmjjJiojnFKyd44

In the preceding program, there's a function that takes a function pointer. Then we call it with two arguments: the first one uses `baz` which is a functor that contains necessary conversion operators. Later, we have a call with a lambda. In this case, the compiler performs the required conversions underneath.

Such conversion might be handy when you need to call a C-style function that requires some callback. For example, below, you can find code that calls `qsort` from the C Library and uses a lambda to sort elements in the reverse order:

```
#include <stdio.h>
#include <stdlib.h>

int main () {
    int values[] = { 8, 9, 2, 5, 1, 4, 7, 3, 6 };
    const size_t numElements = sizeof(values)/sizeof(values[0]);

    qsort(values, numElements, sizeof(int), [](const void * a, const void * b) {
        return ( *(int*)b - *(int*)a );
    });

    for (auto& val : values)
        printf ("%d\n",val);

    return 0;
}
```

You can play with the example @Wandbox[21]

IIFE - Immediately Invoked Function Expression

In most of the examples, you could notice that I defined a lambda and then call it later.

However, you can also invoke lambda immediately:

[21]https://wandbox.org/permlink/xfjO7rLfEPUrjwDT

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

As you can see above, the lambda is created and isn't assigned to any closure object. But then it's called with `()`.

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

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

For example:

```
void BuildStringTest(std::string link, std::string text) {
    const std::string html = [&] {
        const auto& inText = text.empty() ? link : text;
        return "<a href=\"" + link + "\">" + inText + "</a>";
    }(); // call!

    std::cout << html << '\n';
}
```

The above function takes two parameters and then builds a `<a> </a>` HTML tag. Based on the input parameters, we build the `html` variable. If the `text` is not empty, then we use it as the internal HTML value. Otherwise, we use the `link`. We want the `html` variable to be `const`, yet it's hard to write compact code with the required conditions on the input arguments. Thanks to IIFE we can write a separate lambda and then mark our variable with `const`.

One note about the readability

Sometimes having a lambda which is immediately invoked might cause some readability issues.

For example:

```
const auto EnableErrorReporting = [&]() {
    if (HighLevelWarningEnabled())
        return true;

    if (HighLevelWarningEnabled())
        return UsersWantReporting();

    return false;
}();

if (EnableErrorReporting) {
    // ...
}();
```

In the above example, the lambda code is quite complicated, and developers who read the code have to decipher not only that the lambda is invoked immediately, but also they will have to reason about the `EnableErrorReporting` type. They might assume that `EnableErrorReporting` is the closure object and not just a `const` variable. For such cases, you might consider not using `auto` so that we can easily see the type. And maybe even add a comment next to the `}()`, like `// call it now`.

More on IIFE: You may want to read the chapter about C++17 changes and see an upgraded version of IIFE.

Inheriting from a Lambda

It might be surprising to see, but you can also derive from a lambda!

Since the compiler expands a lambda expression into a functor object with `operator()`, then we can derive from that type.

Have a look at the basic code:

```
#include <iostream>

template<typename Callable>
class ComplexFunctor : public Callable
{
public:
    ComplexFunctor(Callable f) : Callable(f) {}
};

template<typename Callable>
ComplexFunctor<Callable> MakeComplexFunctor(Callable&& cal) {
    return ComplexFunctor<Callable>(cal);
}

int main()
{
    auto func = MakeComplexFunctor([]() { std::cout << "Hello Functor!"; });
    func();
}
```

See the code at @Wandbox[22]

In the example, there's the `ComplexFunctor` class which derives from `Callable` which is a template parameter. If we want to derive from a lambda, we need to do a little trick, as we don't know the exact type of the closure type. That's why we need the `MakeComplexFunctor` function that can perform the template argument deduction.

The `ComplexFunctor` apart from its name is just a simple wrapper, without much of a use. Are there any use cases for such code patterns?

For example, We can extend the code above and inherit from two lambdas and create an overloaded set:

[22] https://wandbox.org/permlink/uA4q7Zy1kojUZmqb

```
#include <iostream>

template<typename TCall, typename UCall>
class SimpleOverloaded : public TCall, UCall
{
public:
    SimpleOverloaded(TCall tf, UCall uf) : TCall(tf), UCall(uf) {}

    using TCall::operator();
    using UCall::operator();
};

template<typename TCall, typename UCall>
SimpleOverloaded<TCall, UCall> MakeOverloaded(TCall&& tf, UCall&& uf) {
    return SimpleOverloaded<TCall, UCall>(tf, uf);
}

int main()
{
    auto func = MakeOverloaded(
        [](int) { std::cout << "Int!\n"; },
        [](float) { std::cout << "Float!\n"; }
    );
    func(10);
    func(10.0f);
}
```

See the code @Wandbox[23]

This time we have a bit more code: we derive from two template parameters, but we also need to expose their call operators explicitly.

Why is that? It’s because when looking for the correct function overload the compiler requires the candidates to be in the same scope.

To understand that, let’s write a simple type that derives from two base classes:

[23]https://wandbox.org/permlink/KxuNqP9PO5z3BdVD

```
#include <iostream>

struct BaseInt {
    void Func(int) { std::cout << "BaseInt...\n"; }
};

struct BaseDouble {
    void Func(double) { std::cout << "BaseDouble...\n"; }
};

struct Derived : public BaseInt, BaseDouble {
    //using BaseInt::Func;
    //using BaseDouble::Func;
};

int main() {
    Derived d;
    d.Func(10.0);
}
```

We have two bases classes that implement `Func`. We want to call that method from the derived object.

GCC reports the following error:

```
error: request for member 'Func' is ambiguous
```

See a demo @Wandbox[24]

`::Func()` can be from a scope of `BaseInt` or `BaseDouble`, so the compiler has two scopes to search the best candidate.

Ok, let's go back to our primary use case:

`SimpleOverloaded` is an elementary class, and it's not production-ready. Have a look at the C++17 chapter where we'll discuss an advanced version of this pattern. Thanks to several C++17 features, we'll be able to inherit from multiple lambdas (thanks to variadic templates) and leverage more compact syntax!

[24] https://wandbox.org/permlink/fFRqVGUisdQh1qGV

Summary

In this chapter, you learned how to create and use lambda expressions. I described the syntax, capture clause, type of the lambda, and we covered lots of examples and use cases. We even went a bit further, and I showed you a pattern of deriving from a lambda.

Lambda Expressions become one of the significant parts 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.

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].

What's more, in this chapter, you'll learn about:

- Capturing member variables
- Replacing old functional stuff like `std::bind1st` with modern techniques
- LIFTING

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

Default Parameters for Lambdas

Let's start with some smaller updates:

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

Lambda with Default Parameter

```
#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

If you remember form the previous chapter, the return type for a simple lambda could be deduced by the compiler. This functionality was "extended" into other regular functions and in C++14 you can use `auto` as a return type:

```
auto myFunction() {
    int x = compute(...);
    int y = computeY(...);
    return x + y;
}
```

In the above pseudo-code, the compiler will deduce `int` as a return type.

The concept of deducing return type was improved and extended in C++14. For lambda expressions, it means that they share the same rules as the functions with `auto` return type:

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

[3] https://timsong-cpp.github.io/cppwp/n4140/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][4]

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

```
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 returns `float` while the second `double`. The compiler cannot decide, so you have to select the single type.

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);
```

Since we don't know the type of the closure object, we cannot specify the return type of the above function. The only way was to use `std::function`. But, thanks to the updates in C++14, the compiler can deduce the type for us.

Captures With an Initialiser

Now some more significant updates!

As you recall, in a 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.

[4] https://timsong-cpp.github.io/cppwp/n4140/dcl.spec.auto

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`.

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 expression 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 <algorithm>
#include <iostream>
#include <string>
#include <vector>

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;
```

```
        }
    );
}
```

Play with the example @Wandbox[5]

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

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

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();
}
```

[5] https://wandbox.org/permlink/UCqip5ZpRYeH11WO

Play with code @Wandbox[6]

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:

[6] https://wandbox.org/permlink/Ndl95CEhv8hf0xff

Correct type for map iteration

```
#include <algorithm>
#include <iostream>
#include <map>
#include <string>

int main() {
    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),
    [](const auto& entry) {
        std::cout << entry.first << " = " << entry.second << '\n';
    }
);
```

Now the template argument deduction will adequately get the correct type of the `entry` object and thee will be no additional copy created. Not to mention is the fact that the code is much easier to read and shorter.

You can play with code @Wandbox[7]

[7] https://wandbox.org/permlink/pSbtIA2lgYa6r1bW

Replacing `std::bind1st` and `std::bind2nd` with Lambdas

In the chapter about C++03, I mentioned and showed a few code samples with functional helpers like `std::bind1st` and `std::bind2nd`. However, since C++11 the functionality becomes deprecated, and in C++17, the functions were removed.

Functions like `bind1st()`/`bind2nd()`/`mem_fun()`, … were introduced in the C++98-era and are not needed now as you can apply a lambda or use modern alternatives. What's more, the routines were not updated to handle perfect forwarding, variadic templates, `decltype` and other techniques from C++11. Thus it's best not to use them in modern code.

Here's the list of deprecated functionality:

- `unary_function()`/`pointer_to_unary_function()`
- `binary_function()`/`pointer_to_binary_function()`
- `bind1st()`/`binder1st`
- `bind2nd()`/`binder2nd`
- `ptr_fun()`
- `mem_fun()`
- `mem_fun_ref()`

To replace `bind1st`/`bind2nd` you can use lambdas or `std::bind` (available since C++11) or `std::bind_front` (since C++20).

Let's consider the following code which uses the old functionality:

```
auto onePlus = std::bind1st(std::plus<int>(), 1);
auto minusOne = std::bind2nd(std::minus<int>(), 1);
std::cout << onePlus(10) << ", " << minusOne(10) << '\n';
```

In the preceding example, `onePlus` is a callable object composed of `std::plus` with the first argument fixed. In other words when you write `onePlus(n)` it's then "expanded" into `std::plus(1, n)`.

Similarly, `minusOne` is a composed of `std::minus` with the second argument fixed to one. Thus `minusOne(n)` "expands" into `std::minus(n, 1)`.

The above syntax is quite complicated, so let's see how it can be improved with Modern C++ patterns.

Using Mondern C++ Techniques

Let's try with `std::bind` - which offers more flexibility than `bind1st` or `bind2nd`.

```
using std::placeholders::_1;
auto onePlus = std::bind(std::plus<int>(), _1, 1);
auto minusOne = std::bind(std::minus<int>(), 1, _1);
std::cout << onePlus(10) << ", " << minusOne(10) << '\n';
```

You can play with the code @Compiler Explorer[8]

`std::bind` is more flexible as it can support multiple arguments or cam even reorder them. For argument management, you need to use "placeholders". In our example, we used `_1` to represent the first argument that will be passed to the final function object.

While `std::bind` is much better than the C++03 legacy helpers, it's still not as natural as lambda expressions:

We can write at least two versions. The first one with the hardcoded values for the operations:

```
auto lamOnePlus1 = [](int b) { return 1 + b; };
auto lamMinusOne1 = [](int b) { return b - 1; };
std::cout << lamOnePlus1(10) << ", " << lamMinusOne1(10) << '\n';
```

Still, since C++14 we can also take advantage of capture with initialiser and be more flaxible:

```
auto lamOnePlus = [a=1](int b) { return a + b; };
auto lamMinusOne = [a=1](int b) { return b - a; };
std::cout << lamOnePlus(10) << ", " << lamMinusOne(10) << '\n';
```

The lambda version is much cleaner and more readable.

This will be more visible in a more complicated example below.

Function Composition

As a final example let's have a look at the following code with funcion composition:

[8] https://godbolt.org/z/YeK97m

Funcion composition with std::bind

```
#include <algorithm>
#include <functional>
#include <vector>

int main() {
    using std::placeholders::_1;

    std::vector<int> v { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    const auto val = std::count_if(v.begin(), v.end(),
                                   std::bind(std::logical_and<bool>(),
                                   std::bind(std::greater<int>(), _1, 2),
                                   std::bind(std::less<int>(), _1, 6)));

    return val;
}
```

Can you immediately decipher what's going on there? [9]

You can play with the code @Compiler Explorer[10]

And now let's rewrite this complicated composition with a simple lambda expression:

```
std::vector<int> v { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
const auto more2less6 = std::count_if(v.begin(), v.end(),
                                      [](int x) { return x > 2 && x < 6;});
```

Isn't that better?

You can read more about the guidelines for the use of `std::bind` and lambdas in the following resources: in "Effective Modern C++", Item 34: Prefer lambdas to std::bind, and on the Google Abseil Blog: Tip of the Week #108: Avoid `std::bind`[11].

[9] I used `val` as a vague name on purpose, so its meaning is not clear.
[10] https://godbolt.org/z/0D2cWB
[11] https://abseil.io/tips/108

LIFTing with lambdas

While the algorithms from the Standard Library are convenient, some issues are hard to solve. One of them is passing function overloads into function templates that takes a callable object.

For example:

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);
}
```

In the above example we try to use `foo` which has two overloads for `int` and `float` and pass it into `for_each`. Unfortunately, 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);
                                       ^^^^^
```

The main issue here is that the compiler sees `foo` as a template parameter, so it needs to resolve the type of it. But to do it it would have to check what types `foo` accepts, which is not possible.

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); });
```

Now, we have a wrapper that handles the overload resolution for the compiler.

As you can see, we use value semantics, and the input parameter for the `foo` function will be copied. For more advanced scenarios, this might not be a preferred solution.

If you need a more generic, and better solution then you need to write a bit more code:

```
#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[12]

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 efficiently in template code. In the next chapter, we'll dive into C++17, which brings more updates!

[12] 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`

And in this chapter apart from new features, you'll also learn:

- How to improve the IIFE pattern in C++17
- How to derive from multiple lambdas

Let's start!

[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 initialisation 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 allow 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, one way to make the code safer is to use init capture of `this`[7]:

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

[7] Alternatively, you can also create an overload for the method that works only with references and not rvalue references, disable `auto foo() &&`.

```
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.

Updates To IIFE

In Chapter about C++11 changes you learned about IIFE - Immediately Invoked Functional Expression. In C++17 there's a little update to that technique.

One of the issues with IIFE is that it's sometimes hard to read, as the call operator might be easily skipped when reading the code:

```
const auto var = [&] {
    if (TheFirstCondition())
        return one_value;

    if (TheSecondCindition())
        return second_val;

    return default_value;
}(); // call it!
```

In the C++11 chapter, we even discussed a situation where using `const auto var` might also be a bit misleading. It's because developers might be accustomed to the fact that `var` might be a closure object and not the result of the invocation.

In C++17 there's a handy template function `std::invoke()` that can make IIFE more visible:

```
const auto var = std::invoke([&] {
    if (TheFirstCondition())
        return one_value;

    if (TheSecondCindition())
        return second_val;

    return default_value;
});
```

As you can see, there's no need to write `()` at the end of the expression, and it's now clear that the code *invokes* something.

`std::invoke()` is located in the `<functional>` header file.

Deriving from Multiple Lambdas

In the C++11 chapter, you learned about deriving from a lambda expression. While it was interesting to see such a technique, the use cases were limited.

The main issue with the approach was that it supported only a specific number of lambdas. The examples used one or two base classes. But how about using a variable number of base classes, which means a variable number of lambdas?

In C++17 we have a relatively easy pattern for that!

Have a look:

```
template<class... Ts> struct overloaded : Ts... { using Ts::operator()...; };
template<class... Ts> overloaded(Ts...) -> overloaded<Ts...>;
```

As you can see, we need to use variadic templates, as they allow us to specify the variable number of base classes.

Here's one simple example that uses the code:

The Overloaded Pattern

```
#include <iostream>

template<class... Ts> struct overloaded : Ts... { using Ts::operator()...; };
template<class... Ts> overloaded(Ts...) -> overloaded<Ts...>;

int main() {
    auto test = overloaded{
        [](const int& i) { std::cout << "int: " << i << '\n'; },
        [](const float& f) { std::cout << "float: " << f << '\n'; },
        [](const std::string& s) { std::cout << "string: " << s << '\n';}
    };

    test("10.0f");

    return 0;
}
```

You can play with the code @Compiler Explorer[8]

In the above example, we create a `test` object which is composed of three lambdas. Then we can call the object with a parameter, and the correct lambda will be selected, depending on the type of the input parameter.

Let's now have a closer look at the core parts of this pattern.

Those two lines of code benefits from three features available since C++17:

[8]https://godbolt.org/z/Ns8p9c

- Pack expansions in `using` declarations - short and compact syntax with variadic templates.
- Custom template argument deduction rules - that allows converting a list of lambda objects into a list of base classes for the `overloaded` class. (note: not needed in C++20!)
- Extension to aggregate initialisation - before C++17 you couldn't aggregate initialise type that derives from other types.

In the C++11 chapter, we already covered the need for `using` declaration. This is important for bringing the call operators into the same scope of the `overloaded` structure. In C++17 we got a syntax that supports variadic templates, this was not possible in the previous revisions of the language.

Let's now try to understand the remaining two features:

Custom Template Argument Deduction Rules

We derive from lambdas, and then we expose their `operator()` as we saw in the previous section. But how can we create objects of this `overload` type?

As you know, there's no way to know up-front the type of the lambda, as the compiler has to generate some unique type name for each of them. For example, we cannot just write:

```
overload<LambdaType1, LambdaType2> myOverload { ... } // ???
// what is LambdaType1 and LambdaType2 ??
```

The only way that could work would be some `make` function (as template argument deduction works for function templates since like always):

```
template <typename... T>
constexpr auto make_overloader(T&&... t) {
    return overloaded<T...>{std::forward<T>(t)...};
}
```

With template argument deduction rules that were added in C++17, we can simplify the creation of common template types and the `make_overloader` function is not needed.

For example, for simple types, we can write:

```
std::pair strDouble { std::string{"Hello"}, 10.0 };
// strDouble is std::pair<std::string, double>
```

There's also an option to define custom deduction guides. The Standard library uses a lot of them, for example, for `std::array`:

```
template <class T, class... U>
array(T, U...) -> array<T, 1 + sizeof...(U)>;
```

and the above rule allows us to write:

```
array test{1, 2, 3, 4, 5};
// test is std::array<int, 5>
```

For the `overloaded` patter we can write:

```
template<class... Ts> overloaded(Ts...) -> overloaded<Ts...>;
```

Now, we can type

```
overloaded myOverload { [](int) { }, [](double) { } };
```

And the template arguments for `overload` will be correctly deduced. In our case, the compiler will know the types of lambdas so it will

> Checkout the C++20 chapter as in the new Standard, the Class Template Argument Deduction is improved! For the overloaded pattern, it means that we don't have to write custom deduction guides!

Let's now go to the last missing part of the puzzle - aggregate initialisation.

Extension to Aggregate Initialisation

This functionality is relatively straightforward: we can now initialise a type that derives from other types.

As a reminder: from dcl.init.aggr[9]:

> An aggregate is an array or a class with: * no user-provided, explicit, or inherited constructors
> * no private or protected non-static data members * no virtual functions, and * no virtual, private, or protected base classes

For example (sample from the spec draft):

Aggregate Initialisation

```
struct base1 { int b1, b2 = 42; };

struct base2 {
  base2() { b3 = 42; }
  int b3;
};

struct derived : base1, base2 {
   int d;
};

derived d1{{1, 2}, {}, 4};
derived d2{{}, {}, 4};
```

initializes `d1.b1` with `1`, `d1.b2` with `2`, `d1.b3` with `42`, `d1.d` with `4`, and `d2.b1` with `0`, `d2.b2` with `42`, `d2.b3` with `42`, `d2.d` with `4`.

In our case, it has a more significant impact. Because for the overload class, without the aggregate initialisation, we'd had to implement the following constructor:

[9] https://timsong-cpp.github.io/cppwp/n4659/dcl.init.aggr

```
struct overloaded : Fs... {
  template <class ...Ts>
  overloaded(Ts&& ...ts) : Fs{std::forward<Ts>(ts)}...
  {}

  // ...
}
```

It's a lot of code to write, and probably it doesn't cover all of the cases like `noexcept`.

With aggregate initialisation, we "directly" call the constructor of lambda from the base class list, so there's no need to write it and forward arguments to it explicitly.

OK, we covered a lot, but is there any useful example of the overloaded pattern?

As it appears it might be convenient for `std::variant` visitation.

Example with `std::variant` and `std::visit`

Equipped with the knowledge we can use inheritance and the overloaded pattern for something more practical. Have a look at an example with the visitation of `std::variant`:

The Overloaded Pattern with variant and visit

```
#include <iostream>
#include <variant>

template<class... Ts> struct overloaded : Ts... { using Ts::operator()...; };
template<class... Ts> overloaded(Ts...) -> overloaded<Ts...>;

int main()
{
    auto PrintVisitor = [](const auto& t) { std::cout << t << "\n"; };

    std::variant<int, float, std::string> intFloatString { "Hello" };

    std::visit(PrintVisitor, intFloatString);

    std::visit(overloaded{
        [](int& i) { i*= 2; },
        [](float& f) { f*= 2.0f; },
```

```
        [](std::string& s) { s = s + s; }
    }, intFloatString);

    std::visit(PrintVisitor, intFloatString);

    return 0;
}
```

Play with the code @Compiler Explorer[10]

In the code above we create a variant class that can hold integers, floating-point or string values. Later there's a call to `PrintVisitor` which outputs the current value of the variant. Please notice that thanks to the generic lambda, the visitor can support all types (which have the `<<` operator implemented).

Now, we have another call to `std::visit` that creates a visitor in place, with three different lambda expressions - one for each type. In this artificial example, we want to multiply the value by two, and for strings, it means joining the values together.

Summary

In this chapter, you've seen that C++17 joined two essential elements of C++: `constexpr` with lambdas. Now you can use lambdas in `constexpr` context! This is a necessary step towards improved metaprogramming support in the language. We'll see that even more in the next chapter about C++20. What's more, the C++17 Standard also addressed the capturing `this` problem. In the new standard, you can capture `this` by value so that the code can be much safer.

We also had a look at some uses cases for lambdas: IIFE technique and deriving from lambda expressions. Thanks to the various features enabled in C++17, we now have much nicer syntax and more straightforward ways to write efficient code.

[10]https://godbolt.org/z/usBL7m

5. Lambdas in C++20

During the meeting in Prague, in February 2020, the ISO Committee finally approved C++20 Standard and pushed it to the official publication (probably at the end of 2020). The new specification brings a lot of substantial improvements to the language and the Standard Library! Lambda expressions also got a few upgrades.

In this chapter, you'll see:

- What will change in C++20
- What are the new options to capture `this`
- What are template lambdas
- How to improve generic lambdas with concepts
- How to use lambdas with `constexpr` algorithms
- How to make the `overloaded` patter even shorter

A Quick Overview of the Changes

With C++20 we'll get the following features related to lambda expressions:

- 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]
- `constexpr` Algorithms - most importantly P0202[9],P0879[10] and P1645[11]

If you'd like to know more about C++20 you can have a look at this paper, that summarises all the changes: Changes between C++17 and C++20 DIS - P2131[12].

Let's now have a quick look at the changes.

In most of the cases the newly added features "clean-up" lambda syntax. Plus, C++20 adds enhancements that allow us to use lambdas in advanced scenarios.

For example, with P1091[13] you can capture a structured binding:

[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/p0202
[10] https://wg21.link/P0879
[11] https://wg21.link/P1645
[12] https://wg21.link/P2131
[13] https://wg21.link/P1091

Capturing a structured binding in a lambda

```
auto GetParams() {
    return std::tuple { std::string{"Hello World"}, 42, 10.05f };
}
int main()
{
   auto [x, y, z] = GetParams();
   const auto ParamLength = [&x, &y]() { return x.length() + y; }();
   return ParamLength;
}
```

Play with code @Wandbox[14]

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

Warning about implicit `*this` capture

```
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[15]

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

Another improvement that we got in C++20 is improved pack expansion in lambda init-capture.

[14] https://wandbox.org/permlink/yRosU85B0Q9LnwOv
[15] https://wandbox.org/permlink/yRosU85B0Q9LnwOv

```
template <typename ...Args> void call(Args... args) {
    auto ret = [...capturedArgs = std::move(args)](){};
}
```

Previously, before C++20, the code wouldn't compile and to work around this issue, you had to wrap arguments into a separate tuple. You can read about the history of this capture restriction in P0780[16].

After a quick review, let's have a look at more prominent features in C++20 related to 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:

```
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:

[16]https://wg21.link/P0780

```
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[17]

In the above example, the compiler can warn us about the mismatch in the interface of the lambda.

Another important aspect is that in generic lambda, you only have a variable and not its 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[18]):

[17] https://wandbox.org/permlink/gupbJfUfHHQ2y48q
[18] https://wg21.link/P0428

Deducting from generic argument

```
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:

Using template lambda

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

And another important use case is perfect forwarding in a lambda:

```
// C++17
auto ForwardToTestFunc = [](auto&& ...args) {
  // what's the type of `args` ?
  return TestFunc(std::forward<decltype(args)>(args)...);
};
```

Each time you want to access the type of the template argument, you need to use `decltype()`, but with template lambdas there's not need for that:

```
// C++20:
auto ForwardToTestFunc = []<typename ...T>(T&& ...args) {
  return TestFunc(std::forward<T>(args)...); // we have all the types!
};
```

As you can see, template lambdas provide cleaner syntax and better access to types of arguments.

But there's more! You can also use concepts with lambdas! See in the next section.

Concepts and Lambdas

Concepts are a revolutionary approach for writing templates! They allow you to put constraints on template parameters which improve the readability of code, might speed up compilation time and give better error messages.

One simple example:

A custom concept declaration

```
// define a concept:
template <class T>
concept SignedIntegral = std::is_integral_v<T> && std::is_signed_v<T>;

// use:
template <SignedIntegral T>
void signedIntsOnly(T val) { }
```

In the code above we first create a concept that describes types that are signed and integral. Please notice that we can use existing type traits. Later, we use it to define a template function that supports only types that match the concept. Here we don't use `typename T`, but we can refer to the name of a concept.

Ok, but how that's related to lambda expressions?

The key part here is the terse syntax and constrained `auto` template parameter:

Simplifications and terse syntax

Thanks to the terse concept syntax you can also write templates without the `template<typename..>` part.

With unconstrained `auto`:

```
void myTemplateFunc(auto param) { }
```

Or with constrained `auto`:

```
void signedIntsOnly(SignedIntegral auto val) { }
void floatsOnly(std::floating_point auto fp) { }
```

Such syntax is similar to what you could use in generic lambdas from C++14, as right now you can also write:

```
void myTemplateFunction(auto val) { }
```

In other words, for lambdas, we can leverage this terse style and for example put extra restrictions on the generic lambda argument:

```
auto GenLambda = [](SignedIntegral auto param) { return param*param + 42; };
```

As you can see, in the above example, I restricted the `auto param` with the `SignedIntegral` concept. The whole expression is even more readable than template lambda that we discussed in the previous section.

Here's a bit more complicated example, where we can even define a concept of some class interface:

IRenderable concept, with requires keyword

```
template <typename T>
concept IRenderable = requires(T v) {
    {v.render()} -> std::same_as<void>;
    {v.getVertCount()} -> std::convertible_to<size_t>;
};
```

In the above example we define a concept that matches all types with `render()` and `getVertCount()` member functions. We can then use it to write a generic lambda:

Implementations of `IRenderable` concept/interface

```
#include <concepts>
#include <iostream>

struct Circle {
    void render() { std::cout << "drawing circle\n"; }
    size_t getVertCount() const { return 10; };
};

struct Square {
    void render() { std::cout << "drawing square\n"; }
    size_t getVertCount() const { return 4; };
};

int main() {
    auto RenderCaller = [](IRenderable auto &obj) {
        obj.render();
    };
    Circle c;
    RenderCaller(c);
    Square s;
    RenderCaller(s);
}
```

Play with the code @Wandbox[19]

Changes to Stateless Lambdas

You might recall from the chapter about C++11 that lambdas, even stateless, are not default constructible. However, this limitation is lifted in C++20.

That's why, if your lambda doesn't capture anything, then you can write the following code:

[19] https://wandbox.org/permlink/YXLR8D0i12mi0dlF

A stateless lambda

```
#include <set>
#include <string>
#include <iostream>

struct Product {
    std::string _name;
    int _id {0};
    double _price { 0.0};
};

int main() {
    auto nameCmp = [](const auto& a, const auto& b) {
        return a._name < b._name;
    };
    std::set<Product, decltype(nameCmp)> prodSet {
        {"Cup", 10, 100.0}, {"Book", 2, 200.5 },
        {"TV set", 1, 2000 }, {"Pencil", 4, 10.5}
    };

    for (const auto &elem : prodSet)
        std::cout << elem._name << '\n';
}
```

You can play with code @Wandbox[20]

In the preceding example, I declared a set that stores a list of Products. I need a way to compare products, so I passed a stateless lambda that compares their string names.

For example, if you compiled that code with a C++17 flag, then you'd get an error about using a deleted default constructor:

[20] https://wandbox.org/permlink/dtoMFOThZXTKasuN

```
stl_set.h: In constructor
'std::set<_Key, _Compare, _Alloc>...
[with _Key = Product;
      _Compare = main()::<lambda(const auto:1&, const auto:2&)>;
...
stl_set.h:244:29: error: use of deleted function
'main()::<lambda(const auto:1&, const auto:2&)>::<lambda>()'
```

But in C++20 you can store stateless lambdas and even copy them:

Storing a stateless lambda

```
template <typename F>
struct Product {
    int _id {0};
    double _price { 0.0};
    F _predicate;
};

int main() {
    auto idCmp = [](const auto& a) { return a._id != 0; };
    Product p { 10, 10.0, idCmp };
    [[maybe_unused]] auto p2 = p;
}
```

Play with code @Wandbox[21]

Even more with unevaluated contexts

There are also changes related to advanced uses cases like unevaluated contexts. All together with default constructible lambdas you can now write:

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

As you can see, it's now possible to specify the lambda inside the declaration of map container. It can be used as a comparator functor. Such "unevaluated contexts" are especially handy for advanced template metaprogramming. For example, in the proposal of the feature,

[21] https://wandbox.org/permlink/wTMFVluKdDbsLyOK

the authors mention sorting of tuple objects at compile time using a predicate which is a lambda.

More reasoning in P0315R2[22].

Lambdas and `constexpr` Algorithms

If you recall from the previous chapter since C++17, we can use lambdas which are `constexpr`. With this functionality, you can pass a lambda to functions which are evaluated at compile time. In C++20 most of the standard algorithms are now marked with the `constexpr` keyword which makes `constexpr` lambdas even more convenient!

Let's consider a few examples.

Below you can find code that runs `std::accumulate` on an array, with a custom lambda:

Using std::accumulate with a custom constexpr lambda

```
#include <array>
#include <numeric>

int main() {
    constexpr std::array arr{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    // with constexpr lambda
    static_assert(std::accumulate(begin(arr), end(arr), 0,
        [](auto a, auto b)  {
            return a + b;
        }) == 55);
    return arr[0];
}
```

You can play with code @Compiler Explorer[23]

In the example with `std::accumulate` we used lambda, which is, in fact, the `std::plus` operation.

And in the next example there's a `constexpr` function that takes a `cmp` comparator/predicate for the `count_if` algorithm:

[22]https://wg21.link/P0315R2
[23]https://godbolt.org/z/Tqkphs

Passing constexpr lambda to a custom function

```
#include <array>
#include <algorithm>

constexpr auto CountValues(auto container, auto cmp) {
    return std::count_if(begin(container), end(container), cmp);
}

int main() {
    constexpr auto minVal = CountValues(std::array{-10, 6, 8, 4, -5, 2, 4, 6 },
        [](auto a) { return a >= 0; }
    );
    return minVal;
}
```

Play with code @Compiler Explorer[24]

> What standard algorithms are `constexpr`? All of the algorithms from the `<algorithm>`, `<utility>` and `<numeric>`headers are now marked with `constexpr` except of functions `shuffle`, `sample`, `stable_sort`, `stable_partition`, `inplace_merge` and functions or overloads that accepts the Execution Policy argument. Read more in Papers P0202[25],P0879[26] and P1645[27].

C++20 Updates to the Overloaded Pattern

In the previous chapter, you learned about deriving from multiple lambda expressions and exposing them through the `overloaded` pattern. Such a technique is handy for `std::variant` visitation.

Thanks to the Class Template Argument Deduction (CTAD) updates in C++20 we can now have even shorter syntax!

Why?

[24]https://godbolt.org/z/ouJ_4q
[25]https://wg21.link/p0202
[26]https://wg21.link/P0879
[27]https://wg21.link/P1645

It's because in C++20 there are extensions to CTAD and aggregates are automatically handled. That means that there's no need to write a custom deduction guide.

For a simple type:

```
template <typename T, typename U, typename V>
struct Triple { T t; U u; V v; };
```

In C++20 you can write:

```
Triple ttt{ 10.0f, 90, std::string{"hello"}};
```

And `T` will be deduced as float, `U` as `int` and `V` as `std::string`.

The overloaded pattern in C++20 is now just:

```
template<class... Ts> struct overload : Ts... { using Ts::operator()...; };
```

The proposal for this feature is available in P1021[28] and also P1816[29] (wording).

> ⚠ GCC10 seems to implement this proposal, but it doesn't work for advanced cases with inheritance, so we have to wait for the full conformance here.

Summary

In this chapter, we reviewed the changes that are brought with C++20.

First of all, we have a few clarifications and improvements: for example with the capture of `this`, capturing structured bindings or the ability to default construct stateless lambdas. What's more, there are more significant additions! One of the prominent capabilities now is template lambdas and concepts - so that you get more control over generic lambdas.

To sum up, with C++20 and all of its features, lambdas are even more powerful tools!

[28] https://wg21.link/P1021
[29] https://wg21.link/P1816

Appendix A - List of Techniques

Below you can find a list of techniques and patterns based on lambda expressions used throughout the book.

- Calculating the number of invocations in the C++11 chapter
  – An example of instrumenting a default functor to gather extra information.
- Replacing `std::bind1st`, `std::bind2nd` and removed functional stuff in the C++14 chapter
  – Functions like `std::bind1st`, `std::bind2nd` and other were deprecated in C++11 and removed in C++17. Lambdas might be a good alternative for them.
- Deriving from lambda in the C++11 chapter
  – A basic technique that allows you to wrap a closure type and extend it with additional functionality.
- The overload pattern in the C++17 chapter
  – The mechanism that allows to derive from multiple lambda expressions and pass it to `std::visit`.
- IIFE - Immediately Invoked Function Expression in the C++11 chapter and improvements in the C++17 chapter
  – An efficient way to compute the value of a `const` variable which requires a complex initialisation.
- Passing C++ captureless lambda as a function pointer to C API
  – Lambdas might also be used with C-style API
- An optimisation thanks to capture with initialiser in the C++14 chapter
  – An example of storing a temporary value used for the body of the lambda.
- LIFTING with lambdas in the C++14 chapter
  – This allows passing a set of function overloads into a function template which takes a callable object. For example, when you call algorithms from the Standard Library.

Appendix B - Top Five Advantages of C++ Lambda Expressions[30]

I hope you enjoyed the book and learned a lot about lambda expressions. It appears that this powerful feature has become one of the most visible trademarks of Modern C++. Additionally, the evolution of lambdas is also tightly coupled with the improvements in the language and thus by reading this book, you've also seen a lot of cool C++ techniques.

As a summary for the book, let's wrap our knowledge and list a few benefits of lambdas.

1. Lambdas Make Code More Readable

The first point might sound quite obvious, but it's always good to appreciate the fact that since C++11, we can write more compact code.

For example, in the chapter about C++03 we tried to decipher the following code that used bind expressions and predefined helper functors from the Standard Library:

Functional Composition and bind

```
#include <algorithm>
#include <functional>
#include <vector>

int main() {
    using std::placeholders::_1;
    const std::vector<int> v { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    const auto val = std::count_if(v.begin(), v.end(),
                               std::bind(std::logical_and<bool>(),
                               std::bind(std::greater<int>(),_1, 2),
                               std::bind(std::less_equal<int>(),_1,6)));
    return val;
}
```

[30]this appendix is based on a blog article available at https://www.bfilipek.com/2020/05/lambdasadvantages.html

Play with the code @Compiler Explorer[31]

Can you immediately tell what the final value of `val` is?

Let's now rewrite this into lambda expression:

Cleaner Syntax with Lambdas

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

int main() {
    std::vector<int> v { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    const auto val = std::count_if(v.begin(), v.end(),
                        [](int v) { return v > 2 && v <= 6;});
    return val;
}
```

Isn't that better?

Play with the code @Compiler Explorer[32]

Not only we have shorter syntax for the anonymous function object, but we could even reduce one include statement (as there's no need for `<functional>`any more).

In C++03, it was convenient to use predefined helpers to build those callable objects on the fly. They were handy and allowed you even to compose functionalities to get some complex conditions or operations. However, the main issue is the hard-to-learn syntax. You can of course still use them, even with C++17 or C++20 code (and for places where the use of lambdas is not possible), but I guess that their application for complex scenarios is a bit limited now. In most cases, it's far easier to use lambdas.

I bet you can list a lot of examples from your projects where applying lambda expressions made code much cleaner and easier to read.

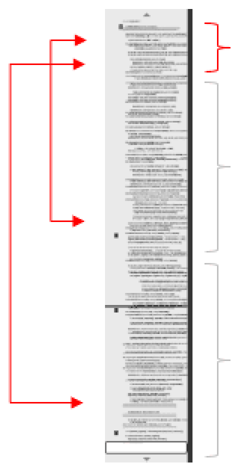
Regarding the readability, we also have another part: locality.

[31]https://godbolt.org/z/_9Ptzg
[32]https://godbolt.org/z/EkjNgK

2. Lambdas Improve Locality of the Code

In C++03, you had to create functions or functors that could be far away from the place where you passed them as callable objects.

This is hard to show on simple artificial examples, but you can imagine a large source file, with more than a thousand lines of code. The code organisation might cause that functors could be located in one place of a file (for example on top). Then the use of a functor could be hundreds of lines further or earlier in the code if you wanted to see the definition of a functor you had to navigate to a completely different place in the file. Such jumping might slow your productivity.



Jumping around a source file

We should also add one more topic to the first and the second point. Lambdas improve locality, readability, but there's also **the naming part**. Since lambdas are anonymous, there's no need for you to select the meaningful name for all of your small functions or functors.

3. Lambdas Allow to Store State Easily

In the C++11 chapter, we covered a simple example of modifying the default comparator for `std::sort` so that we could count the number of invocations.

Capturing state

```
std::vector<int> vec { 0, 5, 2, 9, 7, 6, 1, 3, 4, 8 };

size_t compCounter = 0;
std::sort(vec.begin(), vec.end(), [&compCounter](int a, int b) {
    ++compCounter;
    return a < b;
});
```

Play with the code @Compiler Explorer[33]

As you can see, we can capture a local variable and then use it across all invocations of the binary comparator. Such behaviour is not possible with regular functions (unless you use globals of course), but it's also not straightforward with custom functors types. Lambdas make it very natural and also very convenient to use.

4. Lambdas Allow Several Overloads in the Same Place

This is one of the coolest examples not just related to lambdas, but also to several major Modern C++ features (primarily available in C++17). We learned about this technique in the C++17 chapter, where we discussed the ability to inherit from several lambdas.

Have a look:

The overloaded Pattern

```
template<class... Ts> struct overload : Ts... { using Ts::operator()...; };
template<class... Ts> overload(Ts...) -> overload<Ts...>;

int main() {
    std::variant<int, float, std::string> intFloatString { "Hello" };
    std::visit(overload  {
        [](const int& i) { std::cout << "int: " << i; },
        [](const float& f) { std::cout << "float: " << f; },
        [](const std::string& s) { std::cout << "string: " << s; }
      },
```

[33] https://godbolt.org/z/BgbFWv

```
        intFloatString
    );
}
```

Play with the code @Compiler Explorer[34]

The above example is a handy approach to build a callable object with all possible overloads for `variant` types on the fly. The overloaded pattern is conceptually equivalent to the following structure:

The Print Visitor Structure

```
struct PrintVisitor {
  void operator()(int& i) const { cout << "int: " << i; }
  void operator()(float& f) const { cout << "float: " << f; }
  void operator()(const std::string& s) const { cout << "str: " << s; }
};
```

Additionally, it's also possible to write a compact generic lambda that works for all types from lambda. This can support runtime polymorphism based on `std::variant`.

Runtime Polymorphism Based on std::variant/std::visit

```
#include <variant>

struct Circle { void Draw() const { } };
struct Square { void Draw() const { } };
struct Triangle { void Draw() const { } };

int main() {
    std::variant<Circle, Square, Triangle> shape;
    shape = Triangle{};
    auto callDraw = [](auto& sh) { sh.Draw(); };
    std::visit(callDraw, shape);
}
```

Play with the code @Compiler Explorer[35]

[34] https://godbolt.org/z/fcNdrF
[35] https://godbolt.org/z/EcwqHe

This technique is an alternative to runtime polymorphism based on virtual functions. Here we can work with unrelated types. There's no need for a common base class. You can read about this approach in my blog article at: Runtime Polymorphism with `std::variant` and `std::visit`[36].

5. Lambdas Get Better with Each Revision of C++!

Here's the list of major features related to lambdas that we got with recent C++ Standards:

C++14

- Generic lambdas - you can pass `auto` argument, and then the compiler expands this code into a function template.
- Capture with initialiser - with this feature you can capture not only existing variables from the outer scope, but also create new state variables for lambdas. This also allowed capturing moveable only types.

C++17

- `constexpr` lambdas - in C++17 your lambdas can work in a `constexpr` context.
- Capturing `this` improvements - Before C++17 `this` pointer was captured only as a pointer which might lead to dangling issues. In C++17 you can capture a copy of the object represented by `this*`.

C++20

- Template lambdas - improvements to generic lambdas which offers more control over the input template argument.
- Lambdas and concepts - Lambdas can also work with constrained auto and Concepts, so they are as flexible as functors as template functions
- Lambdas in unevaluated contexts - you can now create a map or a set and use a lambda as a predicate.

[36] https://www.bfilipek.com/2020/04/variant-virtual-polymorphism.html

Your Turn

And what are your favourite features and advantages of lambda expressions? How they simplified your code?

References

- C++11 - [expr.prim.lambda][37]
- C++14 - [expr.prim.lambda][38]
- C++17 - [expr.prim.lambda][39]
- Lambda Expressions in C++ | Microsoft Docs[40]
- Demystifying C++ lambdas - Sticky Bits - Powered by FeabhasSticky Bits – Powered by Feabhas[41]
- The View from Aristeia: Lambdas vs. Closures[42]
- Simon Brand - Passing overload sets to functions[43]
- Jason Turner - C++ Weekly - Ep 128 - C++20's Template Syntax For Lambdas[44]
- Jason Turner - C++ Weekly - Ep 41 - C++17's constexpr Lambda Support[45]

[37] https://timsong-cpp.github.io/cppwp/n3337/expr.prim.lambda
[38] https://timsong-cpp.github.io/cppwp/n4140/expr.prim.lambda
[39] https://timsong-cpp.github.io/cppwp/n4659/expr.prim.lambda
[40] https://docs.microsoft.com/en-us/cpp/cpp/lambda-expressions-in-cpp?view=vs-2017
[41] https://blog.feabhas.com/2014/03/demystifying-c-lambdas/
[42] http://scottmeyers.blogspot.com/2013/05/lambdas-vs-closures.html
[43] https://blog.tartanllama.xyz/passing-overload-sets/
[44] https://www.youtube.com/watch?v=ixGiE4-1GA8&
[45] https://www.youtube.com/watch?v=kmza9U_niq4