C++ Lambda Story

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

From **C++98** to **C++20**

Bartłomiej Filipek

C++ Lambda Story

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

Bartłomiej Filipek

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

This version was published on 2021-07-25

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.

for Wiola and Mikołaj

Contents

About the Book

This book shows the story of lambda expressions in C++. You'll learn how to use this powerful feature in a step-by-step manner, slowly digesting the new capabilities and enhancements that come with each revision of the C++ Standard.

We'll start with C++98/03, and then we'll move on to the latest C++ Standards.

- C++98/03 - 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.
- 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 look at the latest and very fresh C++20 Standard.

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

Walking through the evolution of this powerful C++ feature allows us not only to learn lambdas but also to see how C++ has changed over recent years. In one section you'll see a technique and then it will be "iterated" and updated in further chapters when new C++ elements are available. When possible, the book cross-links to other related sections of the book.

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 (with Tomek's permission, of course :) 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]

Later, 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.

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 almost **four times** 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.

How To Read This Book

This book has the "history" order, so it means that you start from the background behind lambdas, and then you move slowly with new features and capabilities. Reading this book from cover to cover might be suitable for an experienced developer who wants to recall the principles, see the back story and learn what's new in each C++ Standard.

On the other hand, if you are a beginner, it's best to start from the C++11 chapter. See sections about the basic syntax, examples, how to capture variables. Then, when you're ready, you can skip some advanced topics and move into the C++14 chapter where you'll learn about generic lambdas. First parts of the C++11 and C++14 chapter are crucial for understanding lambdas. Once you get the basics, you can read the skipped sections and see more advanced techniques.

At the end of the book in Appendix A, there's a handy list of "lambda techniques". You can have a quick look to see if something is interesting and then start reading that section.

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

Reader Feedback & Errata

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

Here's the errata with the list of fixes:

https://www.cppstories.com/p/cpplambda/

Your feedback matters! If you write an honest review, it can help with the book promotion and the quality of my further work.

If you bought this book through Amazon - as a print or Kindle version - please leave a review there.

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

C++ Lambda Story @GoodReads[3]

Example Code

You can find source code of all examples in this separate Github public repository.

github.com/fenbf/cpplambdastory-code[4]

You can browse individual files or download the whole branch:

github.com/fenbf/cpplambdastory-code/archive/main.zip[5]

Each chapter has its folder, for example, "Lambdas in C++11" has its code in "cpp11".

Each example has a number in the title. For example:

Ex2_3: `std::function` and `auto` type Deduction...

```
// example code...
```

It means that you can go to the second chapter's folder - C++11 and then find the third example. It has the following filename:

`chapter2_cpp11\ex2_3_std_function_and auto.cpp`.

Many examples in the book are relatively short. You can copy and paste the lines into your favourite compiler/IDE and then run the code snippet.

[3] https://www.goodreads.com/book/show/53609731-c-lambda-story
[4] https://github.com/fenbf/cpplambdastory-code
[5] https://github.com/fenbf/cpplambdastory-code/archive/main.zip

Code License

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

Formatting And Special Sections

Code samples are presented in a monospaced font, similar to the following example:

For longer examples:

Title Of the Example

```
#include <iostream>

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

Or shorter snippets (without a title and sometimes `include` statements):

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

When available, you'll also see a link to the online compiler where you can play with the code. For example:

Example title. Live code @Wandbox

```
#include <iostream>

int main() {
    std::cout << "Hello World!";
}
```

You can click on the link in the title and then it should open the website of a given online compiler (in the above case it's Wandbox). You can compile the sample, see the output and experiment with the code directly in your browser.

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

Syntax Highlighting Limitations

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

For example:

- `if constexpr` - Link to Pygments issue: C++ if constexpr not recognized (C++17) · Issue #1136[6].
- The first method of a class is not highlighted - First method of class not highlighted in C++ · Issue #791[7].
- Template method is not highlighted C++ lexer doesn't recognize function if return type is templated · Issue #1138[8].
- Modern C++ attributes are sometimes not recognised properly.

Other issues for C++ and Pygments:C++ Issues · github/pygments/pygments[9].

Special Sections

Throughout the book you can also see the following sections:

This is an Information Box, with extra notes related to the current section.

This is a Warning Box with potential risks and threats related to a given topic.

> This is a Quote Box. In the book, it's often used to quote the C++ Standard.

[6] https://github.com/pygments/pygments/issues/1136
[7] https://github.com/pygments/pygments/issues/791
[8] https://github.com/pygments/pygments/issues/1138
[9] https://github.com/pygments/pygments/issues?q=is%3Aissue+is%3Aopen+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 you edit). They are convenient if you want to play with code samples and check the results using various compilers vendors and versions.

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

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

- Coliru[10] - uses GCC 9.2.0 (as of July 2020), offers link sharing and a basic text editor, it's simple but very effective.
- Wandbox[11] - offers a lot of compilers, including most Clang and GCC versions, can use boost libraries; offers link sharing and multiple file compilation.
- Compiler Explorer[12] - offers many compilers, shows generated assembly code, can execute the code, or even make static code analysis.
- CppBench[13] - runs simple C++ performance tests (using google benchmark library).
- BuildBench[14] - allows to compare build times of two C++ programs, shares a similar UI as CppBench.
- C++ Insights[15] - a Clang-based tool for source to source transformation. It shows how the compiler sees the code by expanding lambdas, auto, structured bindings, template deduction, and variadic packs or range-based for loops.

There's also a helpful list of online compilers gathered on this website: List of Online C++ Compilers[16].

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

About the Author

Bartłomiej (Bartek) Filipek is a C++ software developer from a beautiful city Cracow in Southern Poland. He started his professional career in 2007 and in 2010 he graduated from Jagiellonian University with a Masters Degree in Computer Science.

Bartek currently works at Xara[17], 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[18] and lately at cppstories.com[19]. 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[20]. You can hear Bartek in one @CppCast episode[21] 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[22].

[17] http://www.xara.com/
[18] https://www.bfilipek.com
[19] https://www.cppstories.com]
[20] https://www.meetup.com/C-User-Group-Cracow/
[21] http://cppcast.com/2018/04/bartlomiej-filipek/
[22] https://leanpub.com/cpp17indetail

Acknowledgements

This book wouldn't be possible without valuable input from C++ Expert **Tomasz Kamiński** (see Tomek's profile at Linkedin[23]).

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[24]

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

While the initial version of the book was relatively short, the extended version (additional 100 pages!) was a result of the feedback and encouragement I got from **JFT (John Taylor)**. John spent a lot of time on finding even little things that could be improved and extended.

Also, I'd like to thank **Dawid Pilarski** (panicsoftware.com/about-me[25]) for helpful feedback and a review of the whole book.

Additional words of recognition goes to **Björn Fahller** (@playfulprogramming[26]), **Javier Estrada** (javierestrada.blog[27]) and **Andreas Fertig** (andreasfertig.info[28]) for reviews and extra discussions.

Last but not least, I got a lot of feedback and comments from the blog readers, Patreon Discord Server, and discussions at C++ Polska[29]. Thank you all!

With all of the help from those kind people, the book quality got better and better!

[23] https://www.linkedin.com/in/tomasz-kami%C5%84ski-208572b1/
[24] https://www.meetup.com/pl-PL/C-User-Group-Cracow/events/258795519/
[25] https://blog.panicsoftware.com/about-me/
[26] https://playfulprogramming.blogspot.com/
[27] https://javierestrada.blog/
[28] https://andreasfertig.info/
[29] https://cpp-polska.pl/

Revision History

- 25th March 2019 - The First Edition is live!
- 5th January 2020 - Grammar, better examples, wording, IIFE section, C++20 updates,
- 17th April 2020 - The C++20 chapter is rewritten, grammar, wording, layout,
- 30th April 2020 - Deriving from lambdas, in C++11, C++17 and C++20,
- 19th June 2020 - Major update:
  - Improved C++98/03 chapter, added sections about helper functional objects from the Standard Library,
  - Added a 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 lambda 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,
- 3rd August 2020 - Major Update, also the Kindle Version available:
  - Most code samples have now a link to an online compiler version in the title,
  - Improved description of the syntax of lambdas, showed differences in C++17 and C++20 chapters,
  - New sections: how to store lambdas in a container, Lambdas and Asynchronous Execution, recursive lambdas, Exception Specification in Type System,
  - New section on variadic generic lambdas in C++14 and C++17,
  - New section on variadic packs in C++11, C++20,
  - Use `const` and `noexcept` in longer examples if possible,
  - Lots of smaller changes, improvements, layout across the whole book.
- 30 November 2020 - Corrections, typos, grammar:
  - Wording for data members, function objects (why not a "functor"),
  - Clarification about capturing, initialisation and generated compiler code,
- 1st February 2021 - Print Version of the book!
  - An extended version of the Appendix "Top 6 Lambda Features",
  - Refactoring with IIFE, diagrams for the lambda syntax, index, layout fixes.

1. Lambdas in C++98/03

To start out, it's good to create some background for our main topic. To do this, we'll move into the past and look at code that doesn't use any modern C++ techniques - which means C++98/03 Specification.

In this chapter, you'll learn:

- How to pass function objects to algorithms from the Standard Library in the "old way".
- The limitations of function object class types.
- Why functional helpers weren't good enough.
- The motivation for lambdas for C++0x/C++11.

Callable Objects in C++98/03

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++98/03, this only included function pointers or class types with the call operator (commonly referred as a "functor").

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:

Ex1_1: A basic print function. Live code @Wandbox

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

void PrintFunc(int x) {
    std::cout << x << '\n';
}

int main() {
    std::vector<int> v;
    v.push_back(1); // no uniform initialisation in C++03!
    v.push_back(2); // push_back available only... :)
    std::for_each(v.begin(), v.end(), PrintFunc);
}
```

The code above uses `std::for_each` to iterate over a vector (we use C++98/03 so range-based for loop is not available!) and then it passes `PrintFunc` as a callable object.

We can convert this function into a class type with the call operator:

Ex1_2: A basic print function object type. Live code @Wandbox

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

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

int main() {
    std::vector<int> v;
    v.push_back(1);
    v.push_back(2); // no initialiser list in C++98/03...
    std::for_each(v.begin(), v.end(), Printer());
}
```

The Example defines a `struct` with `operator()` so it means that you can "call" this object like a regular function:

```
Printer printer;
printer();            // calls operator()
printer.operator()(); // equivalent call
```

While non-member functions are usually stateless[1], function-like class types can hold non-static data members which allow storing state. One example is to count the number of invocations of a callable object in an algorithm. This solution needs to keep a counter that is updated with each call:

[1]You can use globals or static variables in a regular function, but it's not the best solution. Such an approach makes it hard to control the state across many groups of lambda invocations.

Ex1_3: Function object with a state. Live code @Wandbox

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

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

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

    int numCalls;
};

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

In the above example, there's a data member `numCalls` which is used to count the number of invocations of the call operator. `std::for_each` returns the function object that we passed it, so we can then take this object and get the data member.

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 data member in our function object and initialise it in the constructor.

Ex1_4: Function object with a 'captured' variable. Live code @Wandbox

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

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

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

    std::string strText;
    int numCalls;
};

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

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

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

What is a "Functor"?

A few sections above I referred that class types with `operator()` are sometimes called "functors". While this term is handy and much shorter than "function object class type" it's not correct.

As it appears, "Functor" comes from functional programming, and it has a different meaning than colloquially used in C++.

Quoting Bartosz Milewski on Functors[2]:

> A functor is a mapping between categories. Given two categories, C and D, a functor F maps objects in C to objects in D — it's a function on objects.

It's very abstract, but fortunately, we can also look at some simpler definition. In chapter 10 of "Functional Programming in C++"[3] Ivan Cukic "translates" those abstract definitions into more practical one for C++:

> A class template `F` is a functor if it has a `transform` (or `map`) function defined on it.

Also, such a `transform` function must obey two rules about identity and composition.

The term "Functor" is not present in any form in the C++ Specification (even in C++98/03), therefore for the rest of this book, we'll try to avoid it.

I recommend the following sources to read more about Functors:

- Functors, Applicatives, And Monads In Pictures - adit.io[4]
- Functors | Bartosz Milewski's Programming Cafe[5]
- What are C++ functors and their uses? - Stack Overflow[6]
- Functor - Wikipedia[7]

[2] https://bartoszmilewski.com/2015/01/20/functors/
[3] "Functional Programming in C++: How to improve your C++ programs using functional techniques 1st Edition" @Amazon
[4] https://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html
[5] https://bartoszmilewski.com/2015/01/20/functors/
[6] https://stackoverflow.com/questions/356950/what-are-c-functors-and-their-uses
[7] https://en.wikipedia.org/wiki/Functor

Issues with Function Object Class Types

As you can see, creating class types with the call operator is very powerful. You have full control, and you can design them any way you like.

However, in C++98/03, the problem was that you had to define a function object type in a different place than the invocation of the algorithm. This could mean that the callable could be dozens or hundreds of lines earlier or further in the source file, or even in a different compilation unit.

As a potential solution, you might have tried writing local classes, since C++ always has support for that syntax. But that didn't work with templates.

See this code:

A Local Function Object Type

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

    std::vector<int> v(10, 1);
    std::for_each(v.begin(), v.end(), LocalPrinter());
}
```

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()::LocalPrinter'
```

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

C++ programmers quickly understood those limitations and found ways to work around the issues with C++98/03. One solution was to prepare a set of helpers. Let's revise them in the next section.

Composing With Functional Helpers

How about having some helpers and predefined function objects?

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.
- `std::mem_fun` - creates a member function wrapper object.
- and many more.

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

Ex1_5: Using old C++98/03 functional helpers. Live code @Wandbox

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

int main() {
    std::vector<int> v;
    v.push_back(1);
    v.push_back(2);
    // .. push back until 9...
    const size_t smaller5 = std::count_if(v.begin(), v.end(),
                                          std::bind2nd(std::less<int>(), 5));
```

```
    return smaller5;
}
```

The example uses `std::less` and fixes its second argument by using `std::bind2nd`. This whole "composition" is passed into `count_if`[8]. 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`.

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:

Ex1_6: Composing functional helpers. Live Code @Wandbox

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

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

    std::vector<int> v;
    v.push_back(1);
    v.push_back(2);
    // push_back until 9...
    const size_t 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;
}
```

[8] `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++98/03 issues. Please use some modern alternatives in your projects. See the C++14 chapter for more information.

The composition uses `std::bind` (from C++11, so we cheated a bit, it's not C++98/03) 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 straightforward to use?

Motivation for a New Feature

As you can see, in C++98/03, 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 function object types, or it was complicated to compose a function with functional helper objects.

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

First of all, the C++ Committee lifted the limitation of the template instantiation with a local type. Since C++11 you can write class types with the call operator locally, in the place where you need them.

What's more, C++11 also brought another idea to life: what if we have a short syntax and then the compiler could "expand" it in a local function object type definition?

And that was the birth of "lambda expressions"!

If we look at N3337[9] - the final draft of C++11, we can see a separate section for lambdas: [expr.prim.lambda][10].

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

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

2. Lambdas in C++11

Hooray! The C++ Committee listened to the opinions of developers, and since C++11 we got lambda expressions!

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

You can read the full specification located under N3337[1] - the final draft of C++11.

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

I think the committee added lambdas in a smart way to the language. They incorporate new syntax, but then the compiler "expands" it into an unnamed "hidden" function object type. This way we have all the advantages (and disadvantages) of the real strongly typed language, and it's relatively easy to reason about the code.

In this chapter, you'll learn:

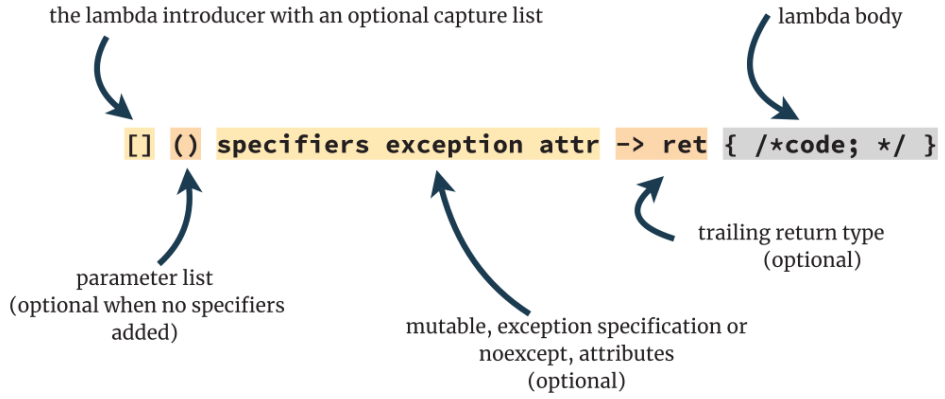
- The basic syntax of lambdas.
- How to capture variables.
- How to capture non-static data members of a class.
- The return type of a lambda.
- What a closure object is.
- How a lambda can be converted to a function pointer and use it with C-style API.
- What's IIFE and why is it useful.
- How to inherit from a lambda expression.

Let's go!

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

The Syntax of Lambda Expression

Below you can find a diagram that illustrates the syntax for lambdas in C++11:



Lambda Syntax in C++11

Let's now see a few examples, just to create some intuition.

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 `[]` section (the lambda introducer) and then the empty `{}` part for the function body. The argument list - `()` - is optional and not needed in this case.

```
// 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; };
[x](int a, int b) mutable noexcept { ++x; return a < b; };
```

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 third lambda uses `mutable` and `noexcept` and they have to appear in that order (you cannot write `noexcept mutable` as the compiler rejects it).

While the `()` part is optional, if you want to apply `mutable` or `noexcept` then `()` needs to be in the expression:

```
// 5. optional ()
[x] { std::cout << x; }; // no () needed
[x] mutable { ++x; };    // won't compile!
[x]() mutable { ++x; };  // fine - () required before mutable
[] noexcept { };         // won't compile!
[]() noexcept { };       // fine
```

The same pattern applies for other specifiers that can be applied on lambdas like `constexpr` or `consteval` in C++17 and C++20.

After basic examples, we can now try to understand how it works and learn full possibilities of lambda expressions.

Core Definitions

Before we go further, it's handy to bring some core definitions from the C++ Standard:

From [expr.prim.lambda#2][3]:

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

As a side note, a lambda expression is a prvalue which is "pure rvalue". This kind of expressions usually yields initialisations and appear on the right-hand side of the assignment (or in a return statement). Read more on C++ Reference[4].

And another definition 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**.

Compiler Expansion

From the above definitions, we can understand that the compiler generates some unique closure type from a lambda expression. Then we can have an instance of this type through the closure object.

Here's a basic example that shows how to write a lambda expression and pass it to `std::for_each`. For comparison, the code also illustrates the corresponding function object type generated by the compiler:

[3] https://timsong-cpp.github.io/cppwp/n3337/expr.prim.lambda#2
[4] https://en.cppreference.com/w/cpp/language/value_category
[5] https://timsong-cpp.github.io/cppwp/n3337/expr.prim.lambda#3

Ex2_1: Lambda and a Corresponding Function Object. Live code @WandBox

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

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

    const std::vector<int> v { 1, 2, 3 };
    std::for_each(v.cbegin(), v.cend(), someInstance);
    std::for_each(v.cbegin(), v.cend(), [] (int x) {
            std::cout << x << '\n';
        }
    );
}
```

In the example, the compiler transforms…

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

…into an anonymous function object that in a simplified form can look as follows:

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

The process of translation or "expansion" can be easily viewed on C++ Insights[6] an online tool which takes valid C++ code and then produces a source code version that the compiler generates: like anonymous function objects for lambdas, template instantiation and many other C++ features.

In the next sections we'll dive more into the individual parts of the lambda expression.

[6] https://cppinsights.io/

The Type of a Lambda Expression

Since the compiler generates a unique name for each lambda (the closure type), there's no way to "spell" it upfront.

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

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

What's more, if you have two lambdas that look the same:

```
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.

We can prove this property with the following code:

Ex2_1: Different Types, Same Code. Live code @Wandbox

```
#include <type_traits>

int main() {
    const auto oneLam = [](int x) noexcept { return x * 2; };
    const auto twoLam = [](int x) noexcept { return x * 2; };
    static_assert(!std::is_same<decltype(oneLam), decltype(twoLam)>::value,
                  "must be different!");
}
```

The example above verifies if the closure types for `oneLam` and `twoLam` are not the same.

In C++17 we can use `static_assert` with no message and also helper variable templates for type traits `is_same_v`:

```
static_assert(std::is_same_v<double, decltype(baz(10))>);
```

However, while you don't know the exact name, you can spell out the signature of the lambda and then store it in `std::function`. In general, what can't be done with a lambda defined as

auto can be done if the lambda is "expressed" through std::function<> type. For example, the previous lambda has a signature of double(int) as it takes an int as an input parameter and returns double. We can then create a std::function object in the following way:

```
std::function<double(int)> myFunc = [](int a) -> double { return 2.0 * a; };
```

std::function is a heavy object because it needs to handle all callable objects. To do that, it requires advanced internal mechanics like type punning or even dynamic memory allocation. We can check its size in a simple experiment:

Ex2_3: `std::function` and `auto` type Deduction. Live code @Wandbox

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

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

    const std::function<double(int)> myFunc =
        [](int a) noexcept -> double {
            return 2.0 * a;
        };

    std::cout << "sizeof(myLambda) is " << sizeof(myLambda) << '\n';
    std::cout << "sizeof(myFunc) is " << sizeof(myFunc) << '\n';

    return myLambda(10) == myFunc(10);
}
```

On GCC the code will print:

```
sizeof(myLambda) is 1
sizeof(myFunc) is 32
```

Because myLambda is just a stateless lambda, it's also an empty class, without any data member fields, so it's minimal size is only one byte. On the other hand, the std::function

version is much larger - 32 bytes. That's why if you can, rely on the `auto` type deduction to get the smallest possible closure objects.

When we talk about `std::function`, it's also important to mention that this type doesn't support moveable-only closures. You can read more about this issue in the C++14 chapter on moveable types.

Constructors and Copying

The section is available only in the full version of the book.

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, in C++11, it's a `const inline` member function. For instance:

```
auto lam = [](double param) { /* do something*/ };
```

Expands into something similar as:

```
struct __anonymousLambda {
    inline void operator()(double param) const { /* do something */ }
};
```

Let's discuss the consequences of this approach and how can we modify the resulting call operator declaration.

Overloading

One thing that is worth mentioning is that when you define a lambda there's no way to create "overloaded" lambdas taking different arguments. Like:

```
// doesn't compile!
auto lam = [](double param) { /* do something*/ };
auto lam = [](int param) { /* do something*/ };
```

Above, the code won't compile as the compiler cannot translate those two lambdas in a single function object. Additionally you cannot redefine the same variable. On the other hand, it's possible to create a function object type which has two call operators:

```
struct MyFunctionObject {
    inline void operator()(double param) const { /* do something */ }
    inline void operator()(int param) const { /* do something */ }
};
```

`MyFunctionObject` can now work with `double` and `int` arguments. If you want a similar behaviour for lambdas, then you can see the section about inheriting from lambdas in this chapter and also about the overloaded pattern from the C++17 chapter.

Attributes

The syntax for lambdas allows using C++11's attributes in the form of `[[attr_name]]`. However, if you apply an attribute to a lambda, then it applies to the type of the call operator and not to the operator itself. That's why currently (and even in C++20) there are no attributes that make sense to put on a lambda. Most compilers even report an error. If we take a C++17 attribute and try to use it with the expression:

```
auto myLambda = [](int a) [[nodiscard]] { return a * a; };
```

This generates the following error on Clang (see live code @Wandbox[7]):

```
error: 'nodiscard' attribute cannot be applied to types
```

While in theory the lambda syntax is prepared, at the moment there are no applicable attributes.

[7] https://wandbox.org/permlink/3zfzL1NNpPXXgLOx

Other Modifiers

We briefly touched on this topic in the syntax section, but you're not limited to a default declaration of the call operator for a closure type. In C++11 you can add `mutable` or an exception specification.

If possible longer examples of this book try to mark the closure object with `const` and also make the lambda `noexcept`.

You can use those keywords by specifying `mutable` and `noexcept` after the parameter declaration clause:

```
auto myLambda = [](int a) mutable noexcept { /* do something */ }
```

The compiler will expand this code into:

```
struct __anonymousLambda {
    inline void operator()(double param) noexcept { /* do something */ }
};
```

Please notice that the `const` keyword is gone now and the call operator can now change the data members of the lambda.

But what data members? How can we declare a data member of lambdas? See the next section about "capturing" variables:

Captures

The section is available only in the full version of the book.

Generated Code

Across this book, I show a possible compiler-generated code as a `struct` to define a closure class type. However, this is only a simplification - a mental model - and inside the compiler, it might be different.

For example, in Clang, the generated AST (Abstract Syntax Tree) uses `class` to represent a closure. The call operator is defined as `public` while data members are `private`.

That's why you cannot write:

```
int x = 0;
auto lam = [x]() { std::cout << x; };
lam.x = 10; // ??
```

In GCC (or similarly in Clang) you'll get:

```
error: 'struct main()::<lambda()>' has no member named 'x'
```

On the other hand, we have an essential part of the specification which mentions, that captured variables are direct initialised, which is impossible for private members (for our regular classes in code). This means that compilers can do a bit "magic" here and create more efficient code (there's no need to copy variables or even move them).

You can read more about the Lambdas internals in a great blog post by Andreas Fertig (the creator of C++ Insights): Under the covers of C++ lambdas - Part 2: Captures, captures, captures[8].

Capture All or Explicit?

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++"[9] by Scott Meyers: "Avoid default capture modes."

The `mutable` Keyword

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

If you want to change this behaviour, you need to add the `mutable` keyword after the parameter list. This syntax effectively removes the `const` from the call operator declaration in the closure type. If you have a simple lambda expression with a `mutable`:

[8] https://andreasfertig.blog/2020/11/under-the-covers-of-cpp-lambdas-part-2-captures-captures-captures/
[9] "Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14" 1st Edition by Scott Meyers, 2014

```
int x = 1;
auto foo = [x]() mutable { ++x; };
```

It will be “expanded” into the following function object:

```
struct __lambda_x1 {
    void operator()() { ++x; }
    int x;
};
```

As you can see, the call operator can change the value of the member fields.

Ex2_5: Capturing Two Variables by Copy and Mutable. Live code @Wandbox

```
#include <iostream>

int main() {
    const auto print = [](const char* str, int x, int y) {
        std::cout << str << ": " << x << " " << y << '\n';
    };
    int x = 1, y = 1;
    print("in main()", x, y);
    auto foo = [x, y, &print]() mutable {
        ++x;
        ++y;
        print("in foo()", x, y);
    };
    foo();
    print("in main()", x, y);
}
```

Output:

```
in main(): 1 1
in foo(): 2 2
in main(): 1 1
```

In the above example, we can change the values of `x` and `y`. Since those are only the 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 you don't need to apply `mutable` to the lambda to modify the value. This is because the captured data members are references which means you cannot rebound them to a new object anyway, but you can change the referenced values.

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

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

One important thing to notice is that when you apply `mutable`, then you cannot mark your resulting closure object with `const` as it prevents you from invoking the lambda!

```
int x = 10;
const auto lam = [x]() mutable { ++x; }
lam(); // doesn't compile!
```

The last line won't compile as we cannot call a non-`const` member function on a `const` object.

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:

Ex2_6: Invocation Counter. Live code @Compiler Explorer

```
#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) noexcept {
            ++compCounter;
            return a < b;
        }
    );

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

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

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 variable and you use `[=]` in your lambda, you might think that your global object is also captured by value... but it's not. See the code:

Ex2_7: Capturing Globals. Live code @Wandbox

```
#include <iostream>

int global = 10;

int main() {
    std::cout << global << '\n';
    auto foo = [=]() mutable noexcept { ++global; };
    foo();
    std::cout << global << '\n';
    const auto increaseGlobal = []() noexcept { ++global; };
    increaseGlobal();
    std::cout << global << '\n';
    const auto moreIncreaseGlobal = [global]() noexcept { ++global; };
    moreIncreaseGlobal();
    std::cout << global << '\n';
}
```

The above example defines `global` and then uses it with several lambdas defined in the `main()` function. If you run the code, then no matter the way you capture, it will always point to the global object, and no local copies will be created.

It's because only variables with automatic storage duration can be captured. GCC can even report the following warning:

```
warning: capture of variable 'global' with non-automatic
         storage duration
```

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

The Clang compiler is even more helpful, as it generates **an error**:

```
error: 'global' cannot be captured because it does not have
       automatic storage duration
```

See Clang live example @Wandbox[10].

[10] https://wandbox.org/permlink/4V91bkuz8NvHrDDA

Capturing Statics

Similarly to capturing global variables, you'll get the same issues with static objects:

Ex2_8: Capturing Static Variables. Live code @Wandbox

```
#include <iostream>

void bar() {
    static int static_int = 10;
    std::cout << static_int << '\n';
    auto foo = [=]() mutable noexcept{ ++static_int; };
    foo();
    std::cout << static_int << '\n';
    const auto increase = []() noexcept { ++static_int; };
    increase();
    std::cout << static_int << '\n';
    const auto moreIncrease = [static_int]() noexcept { ++static_int; };
    moreIncrease();
    std::cout << static_int << '\n';
}

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

This time we try to capture a static variable and then change its value, but since it has no automatic storage duration, the compiler cannot do it.

The output:

```
10
11
12
13
```

GCC reports a warning when you capture the variable by name `[static_int]` and Clang shows an error.

Capturing a Class Member And the `this` Pointer

Things get a bit more complicated where you're in a class member function, and you want to capture a data member. Since all non-static data members are related to the `this` pointer, it also has to be stored somewhere.

Have a look:

Ex2_9: Error when capturing a data member. Live code @Wandbox

```
#include <iostream>

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

    std::string s;
};

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

The code tries to capture `s` which is a data member. But the compiler will emit the following 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 data members.

We can update the code to:

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

    std::string s;
};
```

There are no compiler errors now.

You can also use `[=]` or `[&]` to capture `this` (they both have the same effect in C++11/14!).

Please notice that we captured `this` by value... to a pointer. That's why you have access to the initial data member, not its copy.

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

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

The code won't compile in C++11/14; it is, however, allowed in C++17.

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?

Ex2_10: Returning a Lambda From a Method

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

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

    std::string s;
};

int main() {
   auto f1 = Baz{"abc"}.foo();
```

```
    auto f2 = Baz{"xyz"}.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[11].

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{"abc"}.foo(); // a dangling reference
```

Play with code @Wandbox[12].

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

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

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. See the "Concurrent Execution Using Lambdas" in the C++17 chapter on page 99.

Moveable-only Objects

The section is available only in the full version of the book.

Preserving Const

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

[11] `std::function` is required in C++11 as there's no return type deduction for functions. This limitation is lifted in C++14.
[12] https://wandbox.org/permlink/FOgbNGoQHOmepBgY

Ex2_11: Preserving `const`. Live code @Wandbox

```
#include <iostream>
#include <type_traits>

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

The above code doesn't compile as the captured variable is constant. Here's a possible generated function object for this example:

```
struct __lambda_x {
    void operator()() { x = 11; /*error!*/ }
    const int x;
};
```

You can also play with this code @CppInsight[13].

Capturing a Parameter Pack

To close off our discussion on the capture clause, we should mention that you can also leverage captures with variadic templates. The compiler expands the pack into a list of non-static data members which might be handy if you want to use lambda in a templated code. For example, here's a code sample that experiments with the captures:

[13] https://cppinsights.io/s/7b2f8b10

Ex2_12: Capturing a Variadic Pack. Live code @Wandbox

```
#include <iostream>
#include <tuple>

template<class... Args>
void captureTest(Args... args) {
    const auto lambda = [args...] {
        const auto tup = std::make_tuple(args...);
        std::cout << "tuple size: " <<
                     std::tuple_size<decltype(tup)>::value << '\n';
        std::cout << "tuple 1st:  " << std::get<0>(tup) << '\n';
    };
    lambda(); // call it
}

int main() {
    captureTest(1, 2, 3, 4);
    captureTest("Hello world", 10.0f);
}
```

After running the code, we'll get the following output:

```
tuple size: 4
tuple 1st:  1
tuple size: 2
tuple 1st:  Hello world
```

This somewhat experimental code shows that you can capture a variadic parameter pack by value (by reference is also possible) and then the pack is "stored" into a tuple object. We then call some helper functions on the tuple to access its data and properties.

You can also use C++Insights to see how the compiler generates the code and expands templates, parameter packs and lambdas into code. See the example here @C++Insight[14].

See the C++14 chapter where it's possible to capture moveable only type and also in the C++20 chapter for improvements on variadic parameter pack.

[14] https://cppinsights.io/s/19d3a45d

Return Type

In most cases, even in C++11, you can skip the return type of the lambda and then the compiler will deduce the typename for you.

As a side note: 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[15].

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

From the defect report we can read the following[16]:

> ..If a *lambda-expression* does not include a *trailing-return-type*, it is as if the *trailing-return-type* denotes the following type:
>
> - if there are no return statements in the compound-statement, or all return statements return either an expression of type void or no expression or braced-init-list, the type `void`;
> - otherwise, if all return statements return an expression and the types of the returned expressions after lvalue-to-rvalue conversion (7.3.2 [conv.lval]), array-to-pointer conversion (7.3.3 [conv.array]), and function-to-pointer conversion (7.3.4 [conv.func]) are the same, that common type;
> - otherwise, the program is ill-formed.

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

Ex2_13: Return Type Deduction. Live code @Wandbox

```
#include <type_traits>

int main() {
    const auto baz = [](int x) noexcept {
        if (x < 20)
            return x * 1.1;
        else
            return x * 2.1;
    };
    static_assert(std::is_same<double, decltype(baz(10))>::value,
                  "has to be the same!");
}
```

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

In C++14 the return type of a lambda will be updated to adapt to the rules of `auto` type deduction for regular functions. See "Return Type Deduction" on page 52. This results in a much simpler definition.

Trailing Return Type Syntax

The section is available only in the full version of the book.

Conversion to a Function Pointer

If your lambda doesn't capture any variables then the compiler can convert it to a regular function pointer. See the following description from the Standard expr.prim.lambda#6[17]:

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

> 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.

To illustrate how a lambda can support such conversion let's consider the following example. It defines a function object `baz` that explicitly defines the conversion operator:

Ex2_15: Conversion to a Function Pointer. Live code @Wandboxline-numbers=on

```
#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 << '\n'; };
    } baz;

    callWith10(baz);
    callWith10([](int x) { std::cout << x << '\n'; });
}
```

In the preceding program, there's a function `callWith10` that takes a function pointer. Then we call it with two arguments (lines 18 and 19): the first one uses `baz` which is a function object type that contains necessary conversion operator - it converts to `f_ptr` which is the same as the input parameter for `callWith10`. 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:

Ex2_16: Calling a C-style function. Live code @Wandbox

```
#include <cstdlib>
#include <iostream>

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

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

    for (const auto& val : values)
        std::cout << val << ", ";
}
```

As you can see in the code sample uses `std::qsort` which takes only function pointers as the comparator. The compiler can do an implicit conversion of the stateless lambda that we pass.

A Tricky Case

The section is available only in the full version of the book.

IIFE - Immediately Invoked Functional Expression

In most of examples you've seen so far, you can notice that I defined a lambda and then called it later.

However, you can also invoke a lambda immediately:

Ex2_19: Calling Lambda Immediately. Live code @Wandbox

```
#include <iostream>

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

As you can see above, the lambda is created and isn't assigned to any closure object. But then it's called with `()`. If you run the program, you can expect to see `2, 2` as the output.

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

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

Above, `val` is a constant value of a type returned by a lambda expression, i.e.:

```
// val1 is int
const auto val1 = []() { return 10; }();

// val2 is std::string
const auto val2 = []() -> std::string { return "ABC"; }();
```

Below you can find a longer example where we use IIFE as a helper lambda to create a constant value inside a function:

Ex2_20: IIFE and HTML Generation. Live code @Wandbox

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

void ValidateHTML(const std::string&) { }

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

    ValidateHTML(html);

    return html;
}

int main() {
    try {
        const auto ahref = BuildAHref("www.leanpub.com", "Leanpub Store");
        std::cout << ahref;
    }
    catch (...) {
        std::cout << "bad format...";
    }
}
```

The above example contains a function `BuildAHref` which 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`. Later the variable can be passed to `ValidateHTML`.

One Note About the Readability

The section is available only in the full version of the book.

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 function object with `operator()`, then we can inherit from this type.

Have a look at the basic code:

Ex2_21: Inheriting from a single Lambda. Live code @Wandbox

```
#include <iostream>

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

template<typename Callable>
ComplexFn<Callable> MakeComplexFunctionObject(Callable&& cal) {
    return ComplexFn<Callable>(std::forward<Callable>(cal));
}

int main() {
    const auto func = MakeComplexFunctionObject([]() {
        std::cout << "Hello Complex Function Object!";
    });
    func();
}
```

In the example, there's the `ComplexFn` 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 cannot spell out the exact type of the closure type (unless we wrap it into a `std::function`). That's why we need the `MakeComplexFunctionObject` function that can perform the template argument deduction and get the type of the lambda closure.

The `ComplexFn`, 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:

Ex2_22: Inheriting from two Lambdas. Live code @Wandbox

```
#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>(std::forward<TCall> tf,
                                          std::forward<UCall> uf);
}

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

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. The example also comments out two `using` statements:

Ex2_23: Deriving from two classes, error. Live code @Wandbox

```
#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 base classes which implement `Func`. We want to call that method from the derived object.

GCC reports the following error:

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

Because we commented out the `using` statements `::Func()` can be from a scope of `BaseInt` or `BaseDouble`. The compiler has two scopes to search the best candidate, and according to the Standard, it's not allowed.

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!

Storing Lambdas in a Container

The section is available only in the full version of the book.

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 or storing it in a container.

But that's not all!

Lambda expressions have become a significant part of Modern C++. With more use cases developers also saw possibilities to improve this feature. And that's why you can now move to the next chapter and see essential updates that the ISO Committee added in C++14.

References

C++ Standard Drafts

Here are final drafts of C++ Standards, usually with editorial fixes. Available and hosted at timsong-cpp/cppwp[18].

Sections on Lambda Expression:

- C++11 N3337 - [expr.prim.lambda][19]
- C++14 N4140 - [expr.prim.lambda][20]
- C++17 N4659 - [expr.prim.lambda][21]
- C++20 N4861 - [expr.prim.lambda][22]

Other

- Lambda expressions - cppreference.com[23]
- C++ compiler support - cppreference.com[24]
- Value categories - cppreference.com[25]
- Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14 1st Edition by Scott Meyers, see @Amazon.com[26]
- Functional Programming in C++: How to improve your C++ programs using functional techniques by Ivan Cukic, see @Amazon[27]
- Microsoft Docs - Lambda Expressions in C++[28]

[18] https://github.com/timsong-cpp/cppwp
[19] https://timsong-cpp.github.io/cppwp/n3337/expr.prim.lambda
[20] https://timsong-cpp.github.io/cppwp/n4140/expr.prim.lambda
[21] https://timsong-cpp.github.io/cppwp/n4659/expr.prim.lambda
[22] https://timsong-cpp.github.io/cppwp/n4861/expr.prim.lambda
[23] https://en.cppreference.com/w/cpp/language/lambda
[24] https://en.cppreference.com/w/cpp/compiler_support
[25] https://en.cppreference.com/w/cpp/language/value_category
[26] https://amzn.to/2ZpA7yz
[27] https://amzn.to/3oEMVMY
[28] https://docs.microsoft.com/en-us/cpp/cpp/lambda-expressions-in-cpp?view=vs-2017

- Sticky Bits - Demystifying C++ lambdas[29]
- The View from Aristeia - Lambdas vs. Closures[30]
- Sy Brand - Passing overload sets to functions[31]
- Jason Turner - C++ Weekly - Ep 128 - C++20's Template Syntax For Lambdas[32]
- Jason Turner - C++ Weekly - Ep 41 - C++17's constexpr Lambda Support[33]
- Stack Overflow - c++ - Recursive lambda functions in C++11[34]
- Pedro Melendez - Recursive lambdas in C++(14)[35]
- Andreas Fertig - Under the covers of C++ lambdas - Part 2: Captures, captures, captures[36]
- Scott Meyers - Standard C++ - Universal References in C++11[37]
- Standard C++ Website - Quick Q: Why can noexcept generate faster code than throw()?[38]
- Bjarne Stroustrup - C++ Style and Technique FAQ[39]
- C++ Core Guidelines[40]
- Jonathan Boccara - How Lambdas Make Function Extraction Safer - Fluent C++[41]

[29] https://blog.feabhas.com/2014/03/demystifying-c-lambdas/
[30] http://scottmeyers.blogspot.com/2013/05/lambdas-vs-closures.html
[31] https://blog.tartanllama.xyz/passing-overload-sets/
[32] https://www.youtube.com/watch?v=ixGiE4-1GA8&
[33] https://www.youtube.com/watch?v=kmza9U_niq4
[34] https://stackoverflow.com/questions/2067988/recursive-lambda-functions-in-c11
[35] http://pedromelendez.com/blog/2015/07/16/recursive-lambdas-in-c14/
[36] https://andreasfertig.blog/2020/11/under-the-covers-of-cpp-lambdas-part-2-captures-captures-captures/
[37] https://isocpp.org/blog/2012/11/universal-references-in-c11-scott-meyers
[38] https://isocpp.org/blog/2014/09/noexcept-optimization
[39] https://stroustrup.com/bs_faq2.html
[40] https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines
[41] https://www.fluentcpp.com/2020/11/13/how-lambdas-make-function-extraction-safer/