7.15 — Introduction to lambdas (anonymous functions)

BY NASCARDRIVER ON JANUARY 3RD, 2020 | LAST MODIFIED BY NASCARDRIVER ON FEBRUARY 9TH, 2020

Consider this snippet of code that we introduced in a previous lesson:

```cpp
#include <algorithm>
#include <array>
#include <iostream>
#include <string_view>

static bool containsNut(std::string_view str) // static means internal linkage in this context
{
  // std::string_view::find returns std::string_view::npos if it doesn't find
  // the substring. Otherwise it returns the index where the substring occurs
  // in str.
  return (str.find("nut") != std::string_view::npos);
}

int main()
{
  std::array<std::string_view, 4> arr{ "apple", "banana", "walnut", "lemon" };

  // std::find_if takes a pointer to a function
  auto found{ std::find_if(arr.begin(), arr.end(), containsNut) };

  if (found == arr.end())
  {
    std::cout << "No nuts\n";
  }
  else
  {
    std::cout << "Found " << *found << '\n';
  }

  return 0;
}
```

This code searches through an array of strings looking for the first element that contains the substring “nut”. Thus, it produces the result:

```
Found walnut
```

And while it works, it could be improved.

The root of the issue here is that `std::find_if` requires that we pass it a function pointer. Because of that, we are forced to define a function that’s only going to be used once, that must be given a name, and that must be put in the global scope (because functions can’t be nested!). The function is also so short, it’s almost easier to discern what it does from the one line of code than from the name and comments.

Lambdas to the rescue

A **lambda expression** (also called a **lambda** or **closure**) allows us to define an anonymous function inside another function. The nesting is important, as it allows us both to avoid namespace naming pollution, and to define the function as close to where it is used as possible (providing additional context).

The syntax for lambdas is one of the weirder things in C++, and takes a bit of getting used to. Lambdas take the form:

```
[ captureClause ] ( parameters ) -> returnType
{
    statements;
}
```

The `capture clause` and `parameters` can both be empty if they are not needed.

The `return type` is optional, and if omitted, `auto` will be assumed (thus using type inference used to determine the return type). While we previously noted that type inference for function return types should be avoided, in this context, it's fine to use (because these functions are typically so trivial).

Also note that lambdas have no name, so we don't need to provide one.

As an aside…

This means a trivial lambda definition looks like this:

```
#include <iostream>

int main()
{
  [](){}; // defines a lambda with no captures, no parameters, and no return type

  return 0;
}
```

Let's rewrite the above example using a lambda:

```
#include <algorithm>
#include <array>
#include <iostream>
#include <string_view>

int main()
{
  std::array<std::string_view, 4> arr{ "apple", "banana", "walnut", "lemon" };

  // Define the function right where we use it.
  auto found{ std::find_if(arr.begin(), arr.end(),
                           [](std::string_view str) // here's our lambda, no capture clause
                           {
                             return (str.find("nut") != std::string_view::npos);
                           }) };

  if (found == arr.end())
  {
    std::cout << "No nuts\n";
  }
  else
  {
    std::cout << "Found " << *found << '\n';
  }

  return 0;
}
```

This works just like the function pointer case, and produces an identical result:

```
Found walnut
```

Note how similar our lambda is to our `containsNut` function. They both have identical parameters and function bodies. The lambda has no capture clause (we'll explain what a capture clause is in the next lesson) because it doesn't need one. And we've omitted the trailing return type in the lambda (for conciseness), but since `operator!=` returns a `bool`, our lambda will return a `bool` too.

Type of a lambda

In the above example, we defined a lambda right where it was needed. This use of a lambda is sometimes called a **function literal**.

However, writing a lambda in the same line as it's used can sometimes make code harder to read. Much like we can initialize a variable with a literal value (or a function pointer) for use later, we can also initialize a lambda variable with a lambda definition and then use it later. A named lambda along with a good function name can make code easier to read.

For example, in the following snippet, we're using `std::all_of` to check if all elements of an array are even:

```
// Bad: We have to read the lambda to understand what's happening.
return std::all_of(array.begin(), array.end(), [](int i){ return ((i % 2) == 0); });
```

We can improve the readability of this as follows:

```
// Good: Instead, we can store the lambda in a named variable and pass it to the function.
auto isEven{
  [](int i)
  {
    return ((i % 2) == 0);
  }
};

return std::all_of(array.begin(), array.end(), isEven);
```

Note how well the last line reads: "return whether *all of* the elements in the *array* are *even*"

But what is the type of lambda `isEven`?

As it turns out, lambdas don't have a type that we can explicitly use. When we write a lambda, the compiler generates a unique type just for the lambda that is not exposed to us.

For advanced readers

In actuality, lambdas aren't functions (which is part of how they avoid the limitation of C++ not supporting nested functions). They're a special kind of object called a functor. Functors are objects that contain an overloaded `operator()` that make them callable like a function.

Although we don't know the type of a lambda, there are several ways of storing a lambda for use post-definition. If the lambda doesn't capture anything, we can use a regular function pointer. As soon as the lambda captures anything, a function pointer won't work anymore. However, `std::function` can be used for lambdas even if they are capturing something.

```
#include <functional>

```

```
int main()
{
  // A regular function pointer. The lambda can't capture anything.
  double (*addNumbers1)(double, double){
    [](double a, double b) {
      return (a + b);
    }
  };

  addNumbers1(1, 2);

  // Using std::function. The lambda can capture variables.
  std::function addNumbers2{ // note: pre-C++17, use std::function<double(double, double)> ins
    [](double a, double b) {
      return (a + b);
    }
  };

  addNumbers2(3, 4);

  // Using auto. Stores the lambda with its real type.
  auto addNumbers3{
    [](double a, double b) {
      return (a + b);
    }
  };

  addNumbers3(5, 6);

  return 0;
}
```

The only way of using the lambda's actual type is by means of `auto`. `auto` also has the benefit of having no overhead compared to `std::function`.

Unfortunately, we can't always use `auto`. In cases where the actual lambda is unknown (e.g. because we're passing a lambda to a function as a parameter and the caller determines what lambda will be passed in), we can't use `auto`. In such cases, `std::function` should be used.

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

// We don't know what fn will be. std::function works with regular functions and lambdas.
void repeat(int repetitions, const std::function<void(int)>& fn)
{
  for (int i{ 0 }; i < repetitions; ++i)
  {
    fn(i);
  }
}

int main()
{
  repeat(3, [](int i) {
    std::cout << i << '\n';
  });

  return 0;
}
```

Output

```
0
1
2
```

Rule

Use `auto` when initializing variables with lambdas, and `std::function` if you can't initialize the variable with the lambda.

Generic lambdas

For the most part, lambda parameters work by the same rules as regular function parameters.

One notable exception is that since C++14 we're allowed to use `auto` for parameters (note: in C++20, regular functions will be able to use `auto` for parameters too). When a lambda has one or more `auto` parameter, the compiler will infer what parameter types are needed from the calls to the lambda.

Because lambdas with one or more `auto` parameter can potentially work with a wide variety of types, they are called **generic lambdas**.

For advanced readers

When used in the context of a lambda, `auto` is just a shorthand for a template parameter.

Let's take a look at a generic lambda:

```cpp
#include <algorithm>
#include <array>
#include <iostream>
#include <string_view>

int main()
{
  std::array months{ // pre-C++17 use std::array<std::string_view, 12>
    "January",
    "February",
    "March",
    "April",
    "May",
    "June",
    "July",
    "August",
    "September",
    "October",
    "November",
    "December"
  };

  // Search for two consecutive months that start with the same letter.
  auto sameLetter{ std::adjacent_find(months.begin(), months.end(),
                                      [](const auto& a, const auto& b) {
                                        return (a[0] == b[0]);
                                      }) };

  // Make sure that two months were found.
  if (sameLetter != months.end())
  {
    // std::next returns the next iterator after sameLetter
    std::cout << *sameLetter << " and " << *std::next(sameLetter)
              << " start with the same letter\n";
  }

  return 0;
}
```

Output:

```
June and July start with the same letter
```

In the above example, we use `auto` parameters to capture our strings by `const` reference. Because all string types allow access to their individual characters via `operator[]`, we don't need to care whether the user is passing in a `std::string`, C-style string, or something else. This allows us to write a lambda that could accept any of these, meaning if we change the type of `months` later, we won't have to rewrite the lambda.

However, `auto` isn't always the best choice. Consider:

```
#include <algorithm>
#include <array>
#include <iostream>
#include <string_view>

int main()
{
  std::array months{ // pre-C++17 use std::array<const char*, 12>
    "January",
    "February",
    "March",
    "April",
    "May",
    "June",
    "July",
    "August",
    "September",
    "October",
    "November",
    "December"
  };

  // Count how many months consist of 5 letters
  auto fiveLetterMonths{ std::count_if(months.begin(), months.end(),
                                       [](std::string_view str) {
                                         return (str.length() == 5);
                                       }) };

  std::cout << "There are " << fiveLetterMonths << " months with 5 letters\n";

  return 0;
}
```

Output:

```
There are 2 months with 5 letters
```

In this example, using `auto` would infer a type of `const char*`. C-style strings aren't easy to work with (apart from using `operator[]`). In this case, we prefer to explicitly define the parameter as a `std::string_view`, which allows us to work with the underlying data much more easily (e.g. we can ask the string view for its length, even if the user passed in a C-style array).

Generic lambdas and static variables

One thing to be aware of is that a unique lambda will be generated for each different type that `auto` resolves to. The following example shows how one generic lambda turns into two distinct lambdas:

```cpp
#include <algorithm>
#include <array>
#include <iostream>
#include <string_view>

int main()
{
  // Print a value and count how many times @print has been called.
  auto print{
    [](auto value) {
      static int callCount{ 0 };
      std::cout << callCount++ << ": " << value << '\n';
    }
  };

  print("hello"); // 0: hello
  print("world"); // 1: world

  print(1); // 0: 1
  print(2); // 1: 2

  print("ding dong"); // 2: ding dong

  return 0;
}
```

Output

```
0: hello
1: world
0: 1
1: 2
2: ding dong
```

In the above example, we define a lambda and then call it with two different parameters (a string literal parameter, and an integer parameter). This generates two different versions of the lambda (one with a string literal parameter, and one with an integer parameter).

Most of the time, this is inconsequential. However, note that if the generic lambda uses static duration variables, those variables are not shared between the generated lambdas.

We can see this in the example above, where each type (string literals and integers) has its own unique count! Although we only wrote the lambda once, two lambdas were generated -- and each has its own version of `callCount`.

If we wanted `callCount` to be shared between the lambdas, we'd have to declare it outside of the lambda and capture it by reference so it can be changed by the lambda (we'll discuss captures in the next lesson).

Return type deduction and trailing return types

If return type deduction is used, a lambda's return type is deduced from the `return`-statements inside the lambda. If return type inference is used, all return statements in the lambda must return the same type (otherwise the compiler won't know which one to prefer).

For example:

```
#include <iostream>

int main()
{
  auto divide{ [](int x, int y, bool bInteger) { // note: no specified return type
    if (bInteger)
      return x / y;
    else
      return static_cast<double>(x) / y; // ERROR: return type doesn't match previous return t
  } };

  std::cout << divide(3, 2, true) << '\n';
  std::cout << divide(3, 2, false) << '\n';

  return 0;
}
```

This produces a compile error because the return type of the first return statement (int) doesn't match the return type of the second return statement (double).

In the case where we're returning different types, we have two options:
1) Do explicit casts to make all the return types match, or
2) explicitly specify a return type for the lambda, and let the compiler do implicit conversions.

The second case is usually the better choice:

```
#include <iostream>

int main()
{
  // note: explicitly specifying this returns a double
  auto divide{ [](int x, int y, bool bInteger) -> double {
    if (bInteger)
      return x / y; // will do an implicit conversion to double
    else
      return static_cast<double>(x) / y;
  } };

  std::cout << divide(3, 2, true) << '\n';
  std::cout << divide(3, 2, false) << '\n';

  return 0;
}
```

That way, if you ever decide to change the return type, you (usually) only need to change the lambda's return type, and not touch the lambda body.

Standard library function objects

For common operations (e.g. addition, negation, or comparison) you don't need to write your own lambdas, because the standard library comes with many basic callable objects that can be used instead. These are defined in the **<functional>** header.

In the following example:

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

bool greater(int a, int b)
{
  // Order @a before @b if @a is greater than @b.
  return (a > b);
}

int main()
{
  std::array arr{ 13, 90, 99, 5, 40, 80 };

  // Pass greater to std::sort
  std::sort(arr.begin(), arr.end(), greater);

  for (int i : arr)
  {
    std::cout << i << ' ';
  }

  std::cout << '\n';

  return 0;
}
```

Output

```
99 90 80 40 13 5
```

Instead of converting our `greater` function to a lambda (which would obscure it's meaning a bit), we can instead use `std::greater`:

```
#include <algorithm>
#include <array>
#include <iostream>
#include <functional> // for std::greater

int main()
{
  std::array arr{ 13, 90, 99, 5, 40, 80 };

  // Pass std::greater to std::sort
  std::sort(arr.begin(), arr.end(), std::greater{}); // note: need curly braces to instantiate

  for (int i : arr)
  {
    std::cout << i << ' ';
  }

  std::cout << '\n';

  return 0;
}
```

Output

```
99 90 80 40 13 5
```

Conclusion

Lambdas and the algorithm library may seem unnecessarily complicated when compared to a solution that uses a loop. However, this combination can allow some very powerful operations in just a few lines of code, and can be more readable than writing your own loops. On top of that, the algorithm library features powerful and easy-to-use parallelism, which you won't get with loops. Upgrading source code that uses library functions is easier than upgrading code that uses loops.

Lambdas are great, but they don't replace regular functions for all cases. Prefer a regular functions for non-trivial and reusable cases.

Quiz time

Question #1

Create a `struct Student` that stores the name and points of a student. Create an array of students and use **std::max_element** to find the student with the most points, then print that student's name. `std::max_element` takes the begin and end of a list, and a function that takes 2 parameters and returns `true` if the first argument is less than the second.

Given the following array

```
std::array<Student, 8> arr{
  { { "Albert", 3 },
    { "Ben", 5 },
    { "Christine", 2 },
    { "Dan", 8 }, // Dan has the most points (8).
    { "Enchilada", 4 },
    { "Francis", 1 },
    { "Greg", 3 },
    { "Hagrid", 5 } }
};
```

your program should print

```
Dan is the best student
```

Show Hint

Show Solution

Question #2

Use `std::sort` and a lambda in the following code to sort the seasons by ascending average temperature.

```
#include <algorithm>
#include <array>
#include <iostream>
#include <string_view>

struct Season
{
  std::string_view name{};
  double averageTemperature{};
};

int main()
{
  std::array<Season, 4> seasons{
    { { "Spring", 285.0 },
      { "Summer", 296.0 },
      { "Fall", 288.0 },
      { "Winter", 263.0 } }
  };

  /*
   * Use std::sort here
   */

  for (const auto& season : seasons)
  {
    std::cout << season.name << '\n';
  }

  return 0;
}
```

The program should print

```
Winter
Spring
Fall
Summer
```

Show Solution

7.16 -- Lambda captures

Index

7.14 -- Ellipsis (and why to avoid them)

C++ TUTORIAL | PRINT THIS POST

15 comments to 7.15 — Introduction to lambdas (anonymous functions)

kavin
February 8, 2020 at 8:39 am · Reply

Hi, under generic lambdas example, line 26,

```
return (a[0] == b[0]);
```

return the entire strings "June" and "July" right? And thats why we are derferencing 'sameLetter' in line 32?

In line 32 we can also use *(sameLetter + 1) {from chap 6.8a) instead of *std::next(sameLetter) ? Because i think this is the 1st time we are using std::next() .

nascardriver
February 9, 2020 at 6:52 am · Reply

\`a\` is a string and \`b\` is a string.
\`a[0]\` is a char and \`b[0]\` is a char.
\`a[0] == b[0]\` is a bool.
This is unrelated to the return value of \`std::adjacent_find\`. \`std::adjacent_find\` returns an iterator the the first found element. We use the * operator to get the element an iterator is pointing to.

> we can also use *(sameLetter + 1)
If the container's iterators support the + operator, yes. Not all iterators do that. \`std::next\` works on all iterators than can be advanced forward.

Ged
January 29, 2020 at 8:41 am · Reply

Question 1 In all my codes I have used std::vector instead of std::array. So I have a question. Is std::vector good for built in arrays as well. I'm talking about speed and everything. I keep hearing that the only array that you actually need to know is std::vector. Is this true?

Question 2 [std::begin(vector) and vector.begin()] Are there any real differences between them or just the syntax?

nascardriver
January 30, 2020 at 1:53 am · Reply

1
That's wrong. Here's a good **talk** by Alan Talbot about standard containers. If you know all the values at compile-time, use a type that doesn't need a run-time, ie. `std::array`.

2
`std::begin(vector)` calls `vector.begin()`, there's no difference.

Suyash
January 23, 2020 at 11:58 pm · Reply

The concept of lambda expressions (or anonymous functions) is not a new one for me... And, the syntax is similar to other conventional implementations of this idea in other languages (like Arrow functions in JavaScript)...

What confused me in the above problems was not related to the implementation of the lambdas but some things that I observed that I found peculiar in the given starting code...

Why do we put an extra pair of braces when initializing the fixed-size arrays of Student or Season type in the above code?

(And, yes I did read the answer that was presented in @Alex's link to a Stack Overflow question... But, while it solved some doubts, it raised countless others in its wake...)

If I infer correctly from the answer, then std::array takes in an array and a type as its initialization values but only in the case of user-defined types (classes, struct, enum) but it isn't required for primitive data types(??)... And, only C-style array can be passed as an argument to its constructor or can
we also pass other aggregate types like std::vector or even another std::array?

I would love if you could provide an answer to this burning question or a link to a resource which could ably answer the above queries...

Code in question...

```
// In first question,
std::array<Student, 8> arr{
        {
            { "Albert", 3 },
            { "Ben", 5 },
            { "Christine", 2 },
            { "Dan", 8 },         // Dan has the most points (8).
            { "Enchilada", 4 },
            { "Francis", 1 },
            { "Greg", 3 },
            { "Hagrid", 5 }
        }
    };

// In second question
std::array<Season, 4> seasons{
      { { "Spring", 285.0 },
        { "Summer", 296.0 },
        { "Fall", 288.0 },
        { "Winter", 263.0 } }
    };
```

nascardriver
January 24, 2020 at 2:14 am · Reply

`std::array` doesn't have a custom constructor (nor a custom assignment operator), it's elements (A single c-style array), get initialized via aggregate initialization.
The question about the extra braces is rather common, so I added a **section** about it to lesson S.6.15.
Please let me know if this section helps.

Suyash
January 26, 2020 at 2:21 am · Reply

Yeah, the section does help, especially the example with our own simple implementation of std::array...

Thanks for adding this section... I am sure that it will benefit everyone...

sito
January 22, 2020 at 11:34 am · Reply

hello! I'm a student and I'm currently going through the lessons here. I'm at chapter9 and saw that some new lessons were added. When should i come back to the new lessons? should i continue forward and revisit them at a later date or should i come back to them now before continuing on?

Alex
January 22, 2020 at 11:24 pm · Reply

If the lessons were added earlier in the lesson order than where you are, I'd jump back and read them, as future lessons may have been updated to incorporate those concepts.

Hassan Muhammad
January 22, 2020 at 8:17 am · Reply

finally, i lived to see lambdas, thanks for the hardwork(seems we've two tutors already).
By the way, i've finished the chapter on templates before coming back to this lesson and concerning the auto parameter functions(cpp 20), i though they could substitute templates(because of simplicity), right? or what will you recommend.
keep up the good work and Have a nice day.

nascardriver
January 22, 2020 at 8:28 am · Reply

`auto` parameters are a nice addition, but they can't replace template. Take this simple example

```
void fn(auto a, auto b) {}

template <class T>
void fn2(T a, T b) {}

// ...

fn(1, "hello"); // ok
fn2(1, "hello"); // no, arguments have to have the same type
```

There'll also be concepts in C++20, which need templates.

Connor
January 9, 2020 at 12:49 pm · Reply

Hi!

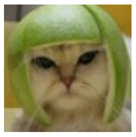
Quick question, and it's probably a really obvious answer that for some reason is not clicking with me, but here we go:

In the quiz questions, when creating our arrays I noticed that there is an extra curly brace and I'm not sure why. When I try to remove it , I get a "too many initializer values" error. But when I create an identical array using C-style arrays, I don't need the extra brace. Maybe I'm just missing something obvious but I thought I'd ask. Thanks :)

Example:

```cpp
std::array<Season, 4> seasons
{
{   // If remove this brace and its closing brace, get error
    {"Spring", 285.0},
    {"Summer", 296.0},
    {"Fall", 288.0},
    {"Winter", 263.0}
}   //
};

Season seasons2[4]   // works okay
{
    {"Spring", 285.0},
    {"Summer", 296.0},
    {"Fall", 288.0},
    {"Winter", 263.0}
}
```



Alex
January 9, 2020 at 1:55 pm · Reply

https://stackoverflow.com/questions/29150369/stdarray-aggregate-initialization-requires-a-confusing-amount-of-curly-braces has some good answers for this.

ZioYuri78
January 6, 2020 at 8:34 am · Reply

Hey guys, super happy to see Lambda chapters!
Just a heads up about the Generic lambdas and static variables example, on mine machine the result is different and looks like it generate a third version of the lambda for print("ding dong") because the callCounter is 0.

0: hello
1: world
0: 1
1: 2
0: ding dong

nascardriver

January 8, 2020 at 5:00 am · Reply

Hi!

Thank you very much for running the code yourself and finding a mistake!
Since `print` takes its parameter by reference, the `auto` turns into a `char` array, not into a `const char*`. "hello" and "world" have the same length, so they share a lambda. "ding dong" is longer, so another lambda is generated. Changing `print`'s parameter to plain `auto` (no reference) fixes it.
I suppose I made the parameter const reference after running the code, without thinking about the consequences.