C++ lambdas

Dr. Yihsiang Liow (April 29, 2024)

Contents

1 Function pointers, functors, and lambdas **3**
1.1 Function as argument and return value 3
1.2 C++ lambda expressions 6
1.3 Lambda expression templates 9
1.4 Lambda types 10
1.5 Lambda captures 11
1.6 When to use lambda expressions 17

Chapter 1

Function pointers, functors, and lambdas

1.1 Function as argument and return value

Suppose you want functions to be **first-class values** (see CISS445), i.e.,

first-class values

1. You want a function to be sent to a function as argument.
2. You want to return a function.

Here's what you can do:

1. Function as argument. This can be done using function pointers or if the function is a functor, i.e., an object with function call operator `operator()`. For function pointer, the function that the function points to must exist. For functor, the class for the functor must exist.
2. Function as return value. Using function pointers, a function pointer can be returned but only if the pointer points to an existing function. For functor, the class for the functor to return must exist.

For function pointers, see CISS245 third pointer notes. For functors, here's a quick review so as to compare against C++ lambdas.

Of course the above is hardcoding the function argument. You should probably do this:

```
#include <iostream>

class Function
{
public:
    virtual int operator()(int x) const = 0;
};

class Square: public Function
{
public:
    int operator()(int x) const
    {
        return x * x;
    }
};

class Cube: public Function
{
public:
    int operator()(int x) const
    {
        return x * x * x;
    }
};

void print(const Function & f, int x)
{
    std::cout << f(x) << '\n';
}

int main()
{
    print(Square(), 3);
    print(Cube(), 3);
    return 0;
}
```

```
[student@localhost cpp-lambda] g++ main.cpp
[student@localhost cpp-lambda] ./a.out
9
27
```

Or even better, do this:

```
#include <iostream>

class Square
{
public:
    int operator()(int x) const
    {
        return x * x;
    }
};
```

```
class Cube
{
public:
    int operator()(int x) const
    {
        return x * x * x;
    }
};

template < typename T >
void print(const T & f, int x)
{
    std::cout << f(x) << '\n';
}

int main()
{
    print(Square(), 3);
    print(Cube(), 3);
    return 0;
}
```

```
[student@localhost cpp-lambda] g++ main.cpp
[student@localhost cpp-lambda] ./a.out
9
27
```

since this will save you from writing an abstract base class. But you still have to write a class for your functor.

1.2 C++ lambda expressions

Try this:

```
#include <iostream>

template < typename T >
void print(const T & f, int x)
{
    std::cout << f(x) << '\n';
}

int main()
{
    print([](int x){ return x * x; }, 3);
    print([](auto x){ return x * x * x; }, 3);
    return 0;
}
```

```
[student@localhost cpp-lambda] g++ main.cpp
[student@localhost cpp-lambda] ./a.out
9
27
```

and this:

```
#include <iostream>

template < typename T, typename S >
void print(const T & f, const S & x)
{
    std::cout << f(x) << '\n';
}

int main()
{
    print([](int x){ return x * x; }, 3);
    print([](auto x){ return x * x * x; }, 3);
    print([](auto x){ return x * x * x; }, 3.1);
    return 0;
}
```

```
[student@localhost cpp-lambda] g++ main.cpp
[student@localhost cpp-lambda] ./a.out
9
27
29.791
```

The

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

is the square function that accepts an integer value for `x` and returns the integer value `x * x`. This is a C++ **lambda expression**. It's said to be an **anonymous function** because this is a function without a name.

lambda expression

anonymous function

Of course you can also give a lambda expression to a name. Try this:

```
#include <iostream>

int main()
{
    auto f = [](int x){ return x * x; };
    std::cout << f(2) << '\n';
    return 0;
}
```

```
[student@localhost cpp-lambda] g++ main.cpp
[student@localhost cpp-lambda] ./a.out
4
```

The return type is deduced. If you want to be explicit, try this:

```
#include <iostream>

int main()
{
    auto f = [](int x) -> int { return x * x; };
    std::cout << f(2) << '\n';
    return 0;
}
```

```
[student@localhost cpp-lambda] g++ main.cpp
[student@localhost cpp-lambda] ./a.out
4
```

A lambda can have no parameter:

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

int main()
{
    auto f = []() { return 42; };
    std::cout << f() << '\n';
    return 0;
}
```

```
[student@localhost cpp-lambda] g++ main.cpp
[student@localhost cpp-lambda] ./a.out
42
```

1.3 Lambda expression templates

If you want your `f` to be a function template, try this:

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

int main()
{
    auto f = [](auto x) { return x * x; };
    std::cout << f(2) << '\n';
    std::cout << f(2.1) << '\n';
    return 0;
}
```

```
[student@localhost cpp-lambda] g++ main.cpp
[student@localhost cpp-lambda] ./a.out
4
4.41
```

If you want to be explicit and include your template type parameter, you can do this:

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

int main()
{
    auto f = []< typename T >(T x) { return x * x; };
    std::cout << f(2) << '\n';
    std::cout << f(2.1) << '\n';
    return 0;
}
```

```
[student@localhost cpp-lambda] g++ main.cpp
[student@localhost cpp-lambda] ./a.out
4
4.41
```

1.4 Lambda types

Here's an example where I explicitly specify the type of of a lambda expression:

```
#include <iostream>
#include <functional> // for std::function

int main()
{
    std::function<int(int)> f = [](int x) -> int { return x * x; };
    std::cout << f(2) << '\n';
    return 0;
}
```

```
[student@localhost cpp-lambda] g++ main.cpp
[student@localhost cpp-lambda] ./a.out
4
```

Here's an example with two parameters:

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

int main()
{
    std::function<int(int, int)> f = [](int x, int y) -> int
                                     { return x + y; };
    std::cout << f(2, 3) << '\n';
    return 0;
}
```

```
[student@localhost cpp-lambda] g++ main.cpp
[student@localhost cpp-lambda] ./a.out
5
```

1.5 Lambda captures

A lambda expression can use variables which are in scope by doing a **capture** (similar to **closure** in CISS445):

capture
closure

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

int main()
{
    int i = 42;
    auto f = [i](int x) { return x + i; };
    std::cout << f(1) << '\n';
    return 0;
}
```

```
[student@localhost cpp-lambda] g++ main.cpp
[student@localhost cpp-lambda] ./a.out
43
```

Now can I change a capture? Try this:

```
#include <iostream>

int main()
{
    int i = 42;
    auto f = [i](int x) { i = 0; return x + i; };
    std::cout << f(1) << '\n';
    return 0;
}
```

```
main.cpp: In lambda function:
main.cpp:6:29: error: assignment of read-only variable i
    6 |     auto f = [i](int x) { i = 0; return x + i; };
      |                           ~~^~~
```

This tells you that captures cannot be changed.

This is really important (see/review closure in CISS445):

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

int main()
{
    int i = 42;
    auto f = [i](int x) { return x + i; };
    std::cout << f(1) << '\n';
    i = 100;
    std::cout << f(1) << '\n';
    return 0;
}
```

```
[student@localhost cpp-lambda] g++ main.cpp
[student@localhost cpp-lambda] ./a.out
43
43
```

Changing `i` in the outer scope does not change the behavior of `f`, i.e., it does not change the captured `i`. This is an example of capture by value, which is the default behavior of the concept of closure in functional programming (CISS445). Compare it with this:

```
#include <iostream>

int main()
{
    int i = 42;
    auto f = [&i](int x) { return x + i; };
    std::cout << f(1) << '\n';
    i = 100;
    std::cout << f(1) << '\n';
    return 0;
}
```

```
[student@localhost cpp-lambda] g++ main.cpp
[student@localhost cpp-lambda] ./a.out
43
101
```

This is called a **capture by reference**. While you cannot modify captures, you can modify captures by reference:

capture by reference

```
#include <iostream>

int main()
{
    int i = 42;
    auto f = [&i](int x) { i = 0; return x + i; };
    std::cout << f(1) << '\n';
    i = 100;
    std::cout << f(1) << '\n';
    return 0;
}
```

```
[student@localhost cpp-lambda] g++ main.cpp
[student@localhost cpp-lambda] ./a.out
1
1
```

To emphasize a capture is not a capture by reference, it's sometimes called a **capture by value**.

capture by value

You can capture by value for multiple variables:

```
#include <iostream>

int main()
{
    int i = 42;
    int j = 1;
    auto f = [i, j](int x) { return x + i + j; };
    std::cout << f(1) << '\n';
    return 0;
}
```

```
[student@localhost cpp-lambda] g++ main.cpp
[student@localhost cpp-lambda] ./a.out
44
```

Or you can capture all by value:

```
#include <iostream>

int main()
{
    int i = 42;
    int j = 1;
    auto f = [=](int x) { return x + i + j; };
    std::cout << f(1) << '\n';
    return 0;
}
```

```
[student@localhost cpp-lambda] g++ main.cpp
[student@localhost cpp-lambda] ./a.out
44
```

You can capture by reference for multiple variables or all variables in scope:

```
#include <iostream>

int main()
{
    int i = 42;
    int j = 1;
    auto f = [&i, &j](int x) { return x + i + j; };
    auto g = [&](int x) { return x + i + j; };
    std::cout << f(1) << '\n';
    std::cout << g(1) << '\n';
    i = 100;
    j = 200;
    std::cout << f(1) << '\n';
    std::cout << g(1) << '\n';
    return 0;
}
```

```
[student@localhost cpp-lambda] g++ main.cpp
[student@localhost cpp-lambda] ./a.out
44
44
301
301
```

Of course remember that your lambda can change variables which are pass by reference (i.e., the captures by reference):

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

int main()
{
    int i = 42;
    int j = 1;
    std::function<int(int)> f = [&i, &j](int x) { i = j = 0; return x; };
    std::cout << f(1) << '\n';
    std::cout << i << ' ' << j << '\n';
    return 0;
}
```

```
[student@localhost cpp-lambda] g++ main.cpp
[student@localhost cpp-lambda] ./a.out
1
0 0
```

In the above, I've shown you how to capture by reference. You cannot capture by constant reference.

Of course when you talk about lambda, after CISS445, you would expect curried functions (unfortunately not as clean as OCAML):

```
#include <iostream>

int main()
{
    auto f = [](int x)
             {
                 return [=](int y) { return x + y; };
             };
    auto g = f(1);
    std::cout << f(1)(2) << '\n';
    std::cout << g(2) << '\n';
    return 0;
}
```

```
[student@localhost cpp-lambda] g++ main.cpp
[student@localhost cpp-lambda] ./a.out
3
3
```

There is a lot more to C++ lambda expressions, which is constantly changing and with more features added every few years.

Here's one more example:

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

int main()
{
    std::vector< int > v {1, 3, 5, 6, 4, 2};

    auto print = [](auto & v)
                 {
                     for (auto e: v) std::cout << e << ' ';
                     std::cout << '\n';
                 };

    std::sort(v.begin(), v.end(), [](int x, int y){ return x < y; });
    print(v);

    std::sort(v.begin(), v.end(), [](int x, int y){ return x > y; });
    print(v);
    return 0;
}
```

```
[student@localhost cpp-lambda] g++ main.cpp
[student@localhost cpp-lambda] ./a.out
1 2 3 4 5 6
6 5 4 3 2 1
```

1.6 When to use lambda expressions

At this point, you should think of C++ lambdas as a cleaner way to write relatively short functors or function that is used only once or twice (to avoid having to write struct or classes and to avoid putting a function name in a global scope).

C++ lambda expressions are nowhere near the power of OCAML functional programming features.

Exercise 1.6.1. Write a recursive fibonacci lambda expression. Answer on the next page.

C++ STL has two sorting algorithms – see comparison-based sorting chapter for details. Here's something from the CISS350 notes:

```
std::vector< int > v0 {5, 3, 1, 4, 2};
std::sort(v0.begin(), v0.end(), std::less< int >());
```

If you want to change the sort order, you just write your own lambda:

```
std::sort(v0.begin(), v0.end(),
          [](auto x, auto y){ return x + y < (x - 1) / y; });
```

If you are writing a function template, for instance a sorting algorithm, you can do this:

```
template < typename T >
void supersort(T * start, T * end)
{
    ...
    if (*p < *q) ...
    ...
}
```

Of course you want a user specify the sort order:

```
template < typename T, typename LESS >
void supersort(T * start, T * end, const LESS & less)
{
    ...
    if (less(*p, *q)) ...
    ...
}
```

Clearly the `less` object has a boolean operator `operator(T*, T*)` that specifies the sort order for the sorting algorithm. You also want the user to default to the standard `std::less` template:

```
template < typename T, typename LESS >
void supersort(T * start, T * end, const LESS & less)
{
    ...
    if (less(*p, *q)) ...
    ...
}
template < typename T >
void supersort(T * start, T * end)
{
    supersort(start, end, std::less< T >());
}
```

After tha above is working, you should change it to use a default template type for the `LESS` type parameter:

```
template < typename T, typename LESS = std::less>
void supersort(T * start, T * end, const LESS & less = LESS())
{
    ...
    if (less(*p, *q)) ...
    ...
}
```

Exercise 1.6.2. Write a mergesort, quicksort and heapsort template.

Here's a recursive fibonacci:

```
#include <iostream>

int main()
{
    auto fib = [](int n)->int
               {
                   auto fib_ = [](auto & self, int n)
                               {
                                   if (n <= 1) return 1;
                                   else
                                       return self(self, n - 1) +
                                              self(self, n - 2);
                               };
                   return fib_(fib_, n);
               };
    std::cout << fib(5) << '\n';

    return 0;
}
```

```
[student@localhost cpp-lambda] g++ main.cpp
[student@localhost cpp-lambda] ./a.out
8
```

Obviously C++'s functional features is lagging way behind OCAML.

Index

anonymous function, 7

capture, 11
capture by reference, 12
capture by value, 13
closure, 11

first-class values, 3

lambda expression, 7