Lambda expressions

Victor Eijkhout, Susan Lindsey

Fall 2022

last formatted: March 27, 2023

1. Why lambda expressions?

Lambda expressions (sometimes incorrectly called 'closures') are 'anonymous functions'. Why are they needed?

- Small functions may be needed; defining them is tedious, would be nice to just write the function recipe in-place.
- C++ can not define a function dynamically, depending on context.
  Example:
  1. we read `float c`
  2. now we want function `float f(float)` that multiplies by c:

```
float c; cin >> c;
float mult( float x ) { // DOES NOT WORK
   // multiply x by c
};
```

2. Introducing: lambda expressions

Traditional function usage:
explicitly define a function and apply it:

```
double sum(float x,float y) { return x+y; }
cout << sum( 1.2, 3.4 );
```

New:
apply the function recipe directly:

Code:

```
[] (float x,float y) -> float {
  return x+y; } ( 1.5, 2.3 )
```

Output:

```
3.8
```

3. Lambda syntax

```
[capture] ( inputs ) -> outtype { definition };
[capture] ( inputs ) { definition };
```

- The square brackets are how you recognize a lambda;
  we will get to the ‘capture’ later.
- Inputs: like function parameters
- Result type specification `->` *`outtype`*:
  can be omitted if compiler can deduce it;
- Definition: function body.

4. Assign lambda expression to variable

Code:

```
auto summing =
  [] (float x,float y) -> float {
  return x+y; };
cout << summing ( 1.5, 2.3 ) << '\n';
cout << summing ( 3.7, 5.2 ) << '\n';
```

Output:

```
3.8
8.9
```

- This is a variable declaration.
- Uses `auto` for technical reasons; see later.

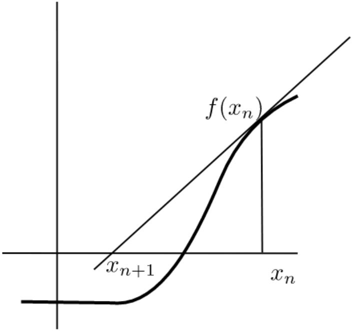
Return type could have been omitted:

```
auto summing =
[] (float x,float y) { return x+y; };
```

Example of lambda usage: Newton's method

5. Newton's method

$$x_{n+1} = x_n - f(x_n)/f'(x_n)$$



6. Newton for root finding

With

$$f(x) = x^2 - 2$$

zero finding is equivalent to

$$f(x) = 0 \quad \text{for } x = \sqrt{2}$$

so we can compute a square root if we have a zero-finding function.

Newton's method for this f:

$$x_{n+1} = x_n - f(x_n)/f'(x_n) = x_n - \frac{(x_n^2 - 2)}{2x_n} = x_n/2 + 2/x_n$$

Square root computation only takes division!

Exercise 1

The Newton method (see HPC book) for finding the zero of a function f, that is, finding the x for which $f(x) = 0$, can be programmed by supplying the function and its derivative:

```
double f(double x) { return x*x-2; };
double fprime(double x) { return 2*x; };
```

and the algorithm:

```
double x{1.};
while ( true ) {
  auto fx = f(x);
  cout << "f( " << x << " ) = " << fx << '\n';
  if (std::abs(fx)<1.e-10 ) break;
  x = x - fx/fprime(x);
}
```

Rewrite this code to use lambda functions for `f` and `fprime`.

You can base this off the file `newton.cxx` *in the repository*

7. Function pointers

You can pass a function to another function.
In C syntax:

```
void f(int i) { /* something with i */ };
void apply_to_5( (void)(*f)(int) ) {
    f(5);
}
int main() {
  apply_to_5(f);
}
```

8. Lambdas as parameter: the problem

Lambdas have a type that is dynamically generated, so you can not write a function that takes a lambda as argument, because you can't write the type.

```
void apply_to_5( /* what? */ f ) {
    f(5);
}
int main() {
    apply_to_5
      ( [] (double x) { cout << x; } );
}
```

(Actually, this simple case does work with C syntax, but not for general lambdas)

9. Lambdas as parameter: the solution

```
#include <functional>
using std::function;
```

With this, you can declare parameters by their signature (that is, types of parameters and output):

Code:

```
void apply_to_5
    ( function< void(int) > f ) {
  f(5);
}
  /* ... */
  apply_to_5
    ( [] (int i) {
      cout << "Int: " << i << '\n';
     } );
```

Output:

```
Int: 5
```

10. Lambdas expressions for Newton

```
#include <functional>
using std::function;
```

With this, you can declare parameters by their signature (that is, types of parameters and output):

```
double newton_root
   ( function< double(double) > f,
     function< double(double) > fprime ) {
```

This states that `f,fprime` are in the class of `double(double)` functions: `double` parameter in, `double` result out.

Exercise 2

Rewrite the Newton exercise above to use a function that is used as:

```
double root = newton_root( f,fprime );
```

Call the function

1. first with the lambda variables you already created;
2. but in a better variant, directly with the lambda expressions as arguments, that is, without assigning them to variables.

Captures

11. Capture parameter

Capture value and reduce number of arguments:

```
int exponent=5;
auto powerfive =
  [exponent] (float x) -> float {
    return pow(x,exponent); };
```

Now `powerfive` is a function of one argument, which computes that argument to a fixed power.

Code:

```
cout << "To the power "
     << exponent << '\n';
for (float x=1.; x<=5.; x+=1.)
  cout << x << ":" << powerfive(x)
     << '\n';
```

Output:

```
To the power 5
1:1
2:32
3:243
4:1024
5:3125
```

12. Capture more than one variable

Example: multiply by a fraction.

```
int d=2,n=3;
times_fraction = [d,n] (int i) ->int {
    return (i*d)/n;
}
```

Exercise 3

- Set two variables

```
float low = .5, high = 1.5;
```

- Define a function of one variable that tests whether that variable is between `low`,`high`.
  (Hint: what is the signature of that function? What is/are input parameter(s) and what is the return result?)

Exercise 4

Extend the newton exercise to compute roots in a loop:

```
for (int n=2; n<=8; n++) {
  cout << "sqrt(" << n << ") = "
       << newton_root(
/* ... */
                      )
       << '\n';
```

Without lambdas, you would define a function

```
double squared_minus_n( double x,int n ) {
  return x*x-n; }
```

However, the `newton_root` function takes a function of only a real argument. Use a capture to make `f` dependent on the integer parameter.

Exercise 5

You don't need the gradient as an explicit function: you can approximate it as

$$f'(x) = \bigl(f(x+h) - f(x)\bigr)/h$$

for some value of h.

Write a version of the root finding function

```
double newton_root( function< double(double)> f )
```

that uses this. You can use a fixed value `h=1e-6`. Do not reimplement the whole newton method: instead create a lambda for the gradient and pass it to the function `newton_root` you coded earlier.

13. Turn it in!

Write a program that

1. reads an integer from the commandline
2. prints a line:
   `The root of this number is 1.4142`
   which contains the word `root` and the value of the square root of the input in default output format.

Your program should

- have a subroutine `newton_root` as described above.
- (8/10 credit): call it with two lambda expressions: one for the function and one for the derivative, *or*
- (10/10 credit) call it with a single lambda expression for the function and approximate the derivative as described above.

The tester is `coe_newton`, options as usual.

More lambda topics

14. Capture by value

Normal capture is by value:

Code:

```
int one=1;
auto increment_by_1 =
  [one] ( int input ) -> int {
    return input+one;
};
cout << increment_by_1 (5)  << '\n';
cout << increment_by_1 (12)  <<
     '\n';
cout << increment_by_1 (25)  <<
     '\n';
```

Output:

```
6
13
26
```

15. Capture by reference

Capture a variable by reference so that you can update it:

```
int count=0;
auto count_if_f =
    [&count] (int i) {
      if (f(i)) count++; }
for ( int i : int_data )
  count_if_f(i);
cout << "We counted: " << count;
```

(See the algorithm header.)

16. Lambdas vs function pointers

Lambda expression with empty capture are compatible with C-style function pointers:

Code:

```
int cfun_add1( int i ) {
  return i+1; };
int apply_to_5( int(*f)(int) ) {
  return f(5); };
//codesnippet  end
  /* ... */
  auto lambda_add1 =
    [] (int i) { return i+1; };
  cout << "C ptr: "
       << apply_to_5(&cfun_add1)
       << '\n';
  cout << "Lambda: "
       << apply_to_5(lambda_add1)
       << '\n';
```

Output:

```
C ptr: 6
Lambda: 6
```

17. Use in algorithms

```
for_each( myarray, [] (int i) { cout << i; } );

transform( myarray, [] (int i) { return i+1; } );
```

See later.