Lambda functions

Victor Eijkhout, Susan Lindsey

Fall 2022

last formatted: September 29, 2022

1. A simple example

You can define a function and apply it:

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

or you can apply the function recipe directly:

Code:

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

Output
[func] lambdadirect:

```
3.8
```

2. 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.

3. Assign lambda to variable

Code:

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

Output

```
[func] lambdavar:

3.8
```

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

Return type could have been ommitted:

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

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:

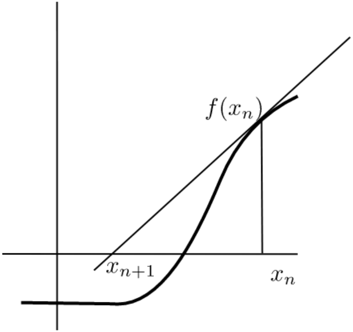
```
double x{1.};
while ( true ) {
  auto fx = f(x);
  cout << "f( " << x << " ) = " << fx << '\n';
  if (std::abs(fx)<1.e-10 ) break;
  x = x - fx/fprime(x);
}
```

Rewrite this code to use lambda functions for f and g.

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

4. Newton's method

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



5. Newton for root finding

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

so

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

Square root computation only takes division!

6. Lambdas as parameter: the problem

Lambdas are in a class that is dynamically generated, so you can not write a function that takes a lambda as argument, because you don't have the class name.

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

(Do not use C-style function pointer syntax.)

7. Lambdas as parameter: the solution

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

With this, you can declare parameters by their signature:

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

This states that `f` is in the class of `double(double)` functions.

Exercise 2

Rewrite the Newton exercise above to use a function with prototype

```
double root = find_zero( f,g );
```

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.

8. 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
[func] lambdait:

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

Exercise 3

Extend the newton exercise to compute roots in a loop:

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

Without lambdas, you would define a function

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

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

Exercise 4

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

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

for some value of h.

Write a version of the root finding function

```
double find_zero( 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 `find_zero` you coded earlier.

9. 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.

10. Capture by value

Normal capture is by value:

```
Code:

int one=1;
auto one_more =
  [one] ( int input ) -> void {
      cout << input+one << '\n';
};
one_more (5);
one_more (6);
one_more (7);
```

```
Output
[func] lambdavalue:

6
7
8
```

11. Capture by value/reference

Capture by reference:

Code:

```
int stride = 1;
auto more_and_more =
  [&stride] ( int input ) -> void {
    cout << input << "=>" <<
     input+stride << '\n';
    stride++;
};
more_and_more(5);
more_and_more(6);
more_and_more(7);
more_and_more(8);
more_and_more(9);
cout << "stride is now: " << stride
     << '\n';
```

```
Output
[func] lambdareference:

5=>6
6=>8
7=>10
8=>12
9=>14
stride is now: 6
```

12. Capture a reduction variable

This mechanism is useful

```
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;
```

13. Lambdas vs 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
[func] lambdacptr:

C ptr: 6
Lambda: 6
```

Lambda in algorithms

14. For each, very simple example

Apply something to each array element:

Code:

```
vector<int>
 ints{2,3,4,5,7,8,13,14,15};
for_each( ints.begin(),ints.end(),
          [] ( int i ) -> void {
            cout << i << '\n';
          }
          );
```

Output
[iter] each:

```
2
3
4
5
7
8
13
14
15
```

15. For any

See if any element satisfies a boolean test:

Code:

```
vector<int>
 ints{2,3,4,5,7,8,13,14,15};
bool there_was_an_8 =
  any_of( ints.begin(),ints.end(),
          [] ( int i ) -> bool {
            return i==8;
          }
          );
cout << "There was an 8: " <<
 boolalpha << there_was_an_8 <<
 '\n';
```

Output
[iter] each:

```
2
3
4
5
7
8
13
14
15
```

16. Capture by reference

Capture variables are normally by value, use ampersand for reference. This is often used in `algorithm` header.

Code:

```
vector<int> moreints{8,9,10,11,12};
int count{0};
for_each
  ( moreints.begin(),moreints.end(),
    [&count] (int x) {
      if (x%2==0)
        count++;
    } );
cout << "number of even: " << count
   << '\n';
```

Output
[stl] counteach:

```
number of even: 3
```

17. For each, with capture

Capture by reference, to update with the array elements.

Code:

```
vector<int>
 ints{2,3,4,5,7,8,13,14,15};
int sum=0;
for_each( ints.begin(),ints.end(),
          [&sum] ( int i ) ->
 void {
            sum += i;
          }
          );
cout << "Sum = " << sum << '\n';
```

Output
[iter] each:

```
2
3
4
5
7
8
13
14
15
```

18. Sorting

Iterator syntax:
(see later for ranges)

```
sort( myvec.begin(),myvec.end() );
```

The comparison used by default is ascending. You can specify other compare functions:

```
sort( myvec.begin(),myvec.end(),
      [] (int i,int j) { return i>j; }
    );
```