

Lambda functions

Victor Eijkhout, Susan Lindsey

Fall 2021

last formatted: October 5, 2021

Lambda functions

1. Lambda example

Write down a 'function recipe' and apply it directly, without creating a 'named function':

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

Store lambda in a variable:

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

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

```
auto f = [] (double x) -> double { return 2*x; };  
y = f(3.7);  
z = f(4.9);
```

- This is a variable declaration.
- Uses `auto` for technical reasons
- See different approach below.

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. 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.)

5. 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 directly with the lambda functions as arguments, that is, without assigning them to variables.

6. 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 << endl;
for (float x=1.; x<=5.; x+=1.)
    cout << x << ":" << powerfive(x) <<
        endl;
```

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 to_the_nth( double x,int 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

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

for some value of h .

Write a version of the root finding function

```
double find_zero( function< double(double)> f, double h=.001 );
```

that uses this. 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.

7. Capture by value/reference

Normal capture is by value:

Code:

```
int one=1;
[one] ( int input ) -> void {
    cout << input+one << endl;
} (5);
```

Output

[func] lambdavalue:

6

Capture by reference:

Code:

```
[&one] ( int input ) -> void {
    one++;
    cout << input+one << endl;
} (5);
cout << "one is now: " << one << endl;
```

Output

[func] lambdareference:

7

one is now: 2

8. 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;
```

9. 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  
     << endl;
```

Output

[stl] counteach:

number of even: 3