Lambda functions

Victor Eijkhout, Susan Lindsey

Fall 2021

last formatted: October 8, 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

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.

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

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

for some value of h.

Write a version of the root finding function

```
double find_zero( function< double(double)> f, 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. 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.

8. 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
```

9. 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;
```

Lambda in algorithms

10. 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
```

11. 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
```

12. 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
```

13. 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
```

Iterators

14. Beyond begin/end

- An iterator is a little like a pointer (into anything iteratable)
- *beginend*
- pointer-arithmetic and 'dereferencing':

```
auto element_ptr = my_vector.begin();
element_ptr++;
cout << *element_ptr;
```

- allows operations (erase, insert) on containers

15. Erase at/between iterators

Erase from start to before-end:

Code:

```
vector<int> counts{1,2,3,4,5,6};
vector<int>::iterator second = counts.
     begin()+1;
auto fourth = second+2; // easier than
      'iterator'
counts.erase(second,fourth);
cout << counts[0] << "," << counts[1]
     << "\n";
```

Output
[iter] erase2:

```
1,4
```

(Also single element without end iterator.)

16. Insert at iterator

Insert at iterator: value, single iterator, or range:

Code:

```
vector<int> counts{1,2,3,4,5,6},zeros
     {0,0};
auto after_one = zeros.begin()+1;
zeros.insert( after_one,counts.begin()
     +1,counts.begin()+3 );
//vector<int>::insert( after_one,
     counts.begin()+1,counts.begin()+3
      );
cout << zeros[0] << "," << zeros[1] <<
      ","
     << zeros[2] << "," << zeros[3]
     << "\n";
```

Output
[iter] insert2:

```
0,2,3,0
```