Lambda expressions

Victor Eijkhout, Susan Lindsey

Fall 2025
last formatted: September 9, 2025

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:

```
1 float c; cin >> c;
2 float mult( float x ) { // DOES NOT WORK
3   // multiply x by c
4 };
```

2. Introducing: lambda expressions

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

```
1 float sum(float x,float y) { return x+y; }
2 // and in main:
3 cout << sum( 1.2f, 3.4f );
```

New:
apply the function recipe directly:

```
  Code:

1 // lambda/lambdaex.cpp
2 [] (float x,float y) -> float {
3   return x+y; } ( 1.5, 2.3 )
```

```
 Output:

1 3.8
```

3. Lambda syntax

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

- The square brackets are how you recognize a lambda;
  we will get to the 'capture' later. For now it will often be
  empty.
- 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**

Lambda expression assigned to a variable:

```
Code:
1 // lambda/lambdaex.cpp
2 auto summing =
3   [] (float x,float y) -> float {
4   return x+y; };
5 cout << summing ( 1.5, 2.3 ) << '\n';
6 cout << summing ( 3.7, 5.2 ) << '\n';
```

```
Output:
1 3.8
2 8.9
```

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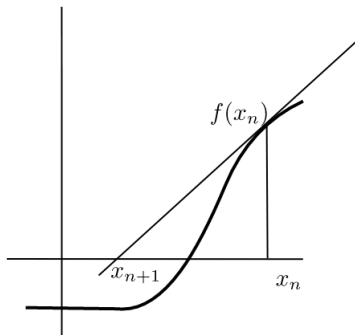- Uses `auto` for technical reasons; see later.

Return type could have been omitted:

```
1   auto summing =
2   [] (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

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

If you use variables for the lambda expressions, put them in the main program.

7. Function pointers

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

```c
1 void f(int i) { /* something with i */ };
2 void apply_to_5( (void)(*f)(int) ) {
3     f(5);
4 }
5 int main() {
6   apply_to_5(f);
7 }
```

8. Lambdas as parameter

We want to write lambda expressions in-line in a function call:

```
1 int main() {
2     apply_to_5
3         ( [] (double x) { cout << x; } );
4 }
```

9. 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.

```
1 void apply_to_5( /* what? */ func ) {
2     func(5);
3 }
4 int main() {
5     apply_to_5
6         ( [] (double x) { cout << x; } );
7 }
```

10. Lambdas as parameter: the solution

Header:

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

declare function parameters by their signature
(that is, types of parameters and output):

```
Code:
1 // lambda/lambdaex.cpp
2 void apply_to_5
3     ( function< void(int) > f ) {
4   f(5);
5 }
6     /* ... */
7   apply_to_5
8     ( [] (int i) {
9       println("Int: {}",i);
10      } );
```

```
Output:
1 Int: 5
```

11. **Lambdas expressions for Newton**

```cpp
1 #include <functional>
2 using std::function;
```

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

```cpp
1 // newton/newton-lambda.cpp
2 double newton_root
3    ( function< double(double) > f,
4      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 by implementing a `newton_root` function:

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

Call the function

1. first with the lambda variables you already created;
2. then directly with the lambda expressions as arguments, that is, without assigning them to variables.

Captures

12. **Capture value is copied**

Illustrating that the capture variable is copied once and for all:

```
Code:
1 // lambda/lambdacapture.cpp
2 int inc;
3 cin >> inc;
4 auto increment =
5    [inc] ( int input ) -> int {
6      return input+inc;
7    };
8 println("increment by: {}",inc);
9 println("1 -> {}",increment(1));
10 inc = 2*inc;
11 println("1 -> {}",increment(1));
```

```
Output:
1 increment by: 2
2 1 -> 3
3 1 -> 3
```

Exercise 3

Write a program that

- reads a `float` factor;
- defines a function `multiply` of one argument that multiplies its input by that factor.

13. **Capture more than one variable**

Example: multiply by a fraction.

```
1 int d=2,n=3;
2 times_fraction = [d,n] (int i) ->int {
3     return (i*d)/n;
4 }
```

Exercise 4

- Set two variables

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

- Define a function *is_in_range* 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 5

Extend the Newton exercise to compute roots in a loop:

```
// newton/newton-lambda.cpp
for ( float n=2.; n<10; n:=.5 ) {
  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.

TACC

Exercise 6

Write a version of the root finding function that only takes the objective function:

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

and approximates the derivative by a finite difference. You can use a fixed value `h=1e-6`.

Do not reimplement the whole newton method: instead create a lambda expression for the gradient and pass it to the function `newton_root` you coded earlier as second argment.

14. **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

15. **Capture by value**

Normal capture is by value:

```
Code:
1 // lambda/lambdacapture.cpp
2 int n;
3 cin >> n;
4 auto increment_by_n =
5    [n] ( int input ) -> int {
6       return input+n;
7 };
8 println("{}",increment_by_n (5));
9 println("{}",increment_by_n (12));
10 println("{}",increment_by_n (25));
```

```
Output:
1 (input value: 1)
2
3 6
4 13
5 26
```

16. **Capture by reference**

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

```
Code:
1 // lambda/countif.cpp
2 int count=0;
3 auto count_if_f =
4   [&count] (int i) {
5     if (i%2==0) count++; };
6 for ( int i : {1,2,3,4,5} )
7   count_if_f(i);
8 println("We counted: {}",count);
```

```
Output:
1 We counted: 2
```

17. **Lambdas vs function pointers**

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

```
Code:
1 // lambda/lambdacptr.cpp
2 int cfun_add1( int i ) {
3   return i+1; };
4 int apply_to_5( int(*f)(int) ) {
5   return f(5); };
6   /* ... */
7   auto lambda_add1 =
8     [] (int i) { return i+1; };
9   cout << "C ptr: "
10        << apply_to_5(&cfun_add1)
11        << '\n';
12  cout << "Lambda: "
13        << apply_to_5(lambda_add1)
14        << '\n';
```

```
Output:
1 /bin/sh: line 0:
      ↪eval: -g:
      ↪invalid option
2 eval: usage: eval
      ↪[arg ...]
3 make[1]: ***
      ↪[lambdacptr.o]
      ↪Error 2
```

18. **Use in algorithms**

```
1    for_each( myarray, [] (int i) { cout << i; } );
2
3    transform( myarray, [] (int i) { return i+1; } );
4
```

See later.