Lambda expressions

Victor Eijkhout, Susan Lindsey

Fall 2025

last formatted: September 11, 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`:

```
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:

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

New:
apply the function recipe directly:

Code:

```
// lambda/lambdaex.cpp
[] (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. 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:

```
// lambda/lambdaex.cpp
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.

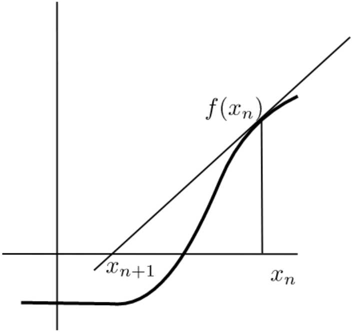
Return type could have been omitted:

```
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

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:

```
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

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

```
int main() {
    apply_to_5
      ( [] (double x) { cout << x; } );
}
```

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.

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

10. Lambdas as parameter: the solution

Header:

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

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

Code:

```
// lambda/lambdaex.cpp
void apply_to_5
    ( function< void(int) > f ) {
  f(5);
}
    /* ... */
  apply_to_5
    ( [] (int i) {
      println("Int: {}",i);
     } );
```

Output:

```
Int: 5
```

11. Lambdas expressions for Newton

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

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

```cpp
// newton/newton-lambda.cpp
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 by implementing a *newton_root* function:

```
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:

```
// lambda/lambdacapture.cpp
int inc;
cin >> inc;
auto increment =
  [inc] ( int input ) -> int {
    return input+inc;
  };
println("increment by: {}",inc);
println("1 -> {}",increment(1));
inc = 2*inc;
println("1 -> {}",increment(1));
```

Output:

```
increment by: 2
1 -> 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.

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

Exercise 4

- Set two variables

```
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.

Exercise 6

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

```
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.

More lambda topics

14. Capture by value

Normal capture is by value:

Code:

```
// lambda/lambdacapture.cpp
int n;
cin >> n;
auto increment_by_n =
  [n] ( int input ) -> int {
    return input+n;
};
println("{}",increment_by_n (5));
println("{}",increment_by_n (12));
println("{}",increment_by_n (25));
```

Output:

```
(input value: 1)

6
13
26
```

15. Capture by reference

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

Code:

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

Output:

```
We counted: 2
```

16. Lambdas vs function pointers

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

Code:

```
// lambda/lambdacptr.cpp
int cfun_add1( int i ) {
  return i+1; };
int apply_to_5( int(*f)(int) ) {
  return f(5); };
    /* ... */
  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:

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

17. Use in algorithms

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

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

See later.