

# Lambda expressions

Victor Eijkhout, Susan Lindsey

Fall 2023

last formatted: September 20, 2023

# 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:

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

New:  
apply the function recipe directly:

Code:

```
1 // func/lambdaex.cpp  
2 [] (float x, float y) -> float {  
3     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

Code:

```
1 // func/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:

3.8  
8.9

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

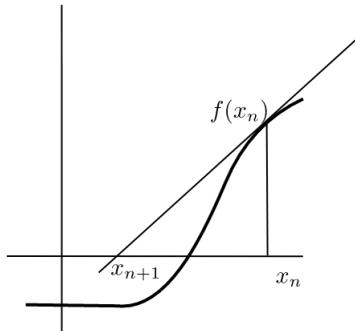
Return type could have been omitted:

```
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  $f_{prime}$ .

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

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

## 7. Function pointers

You can pass a function to another function.

In C syntax:

```
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 }
```

(You don't have to understand this syntax. The point is that you can pass a function as argument.)

## 8. 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? */ f ) {  
2     f(5);  
3 }  
4 int main() {  
5     apply_to_5  
6     ( [] (double x) { cout << x; } );  
7 }
```

## 9. Lambdas as parameter: the solution

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

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

Code:

```
1 // func/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     cout << "Int: " << i << '\n';
10    } );
```

Output:

Int: 5

## 10. Lambdas expressions for Newton

We are going to write a Newton function which takes two parameters: an objective function, and its derivative; it has a `double` as result.

```
// newton/newton-lambda.cpp
double newton_root
( function< double(double) > f,
  function< double(double) > fprime ) {
```

This states that  $f, f_{prime}$  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

# 11. Capture variable

Increment function:

- scalar in, scalar out;
- the increment amount has been fixed through the capture.

Code:

```
1 // func/lambdacapture.cpp
2 int one=1;
3 auto increment_by_n =
4     [one] ( int input ) -> int {
5     return input+one;
6 };
7 cout << increment_by_n (5) << '\n';
8 cout << increment_by_n (12) << '\n';
9 cout << increment_by_n (25) << '\n';
```

Output:

```
6
13
26
```



## 12. 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 3

- Set two variables

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

- Define a function 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 4

Extend the newton exercise to compute roots in a loop:

```
// newton/newton-lambda.cpp
for (int n=2; n<=8; ++n) {
    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.

## 13. Derivative by finite difference

The Newton method also works if the derivative is not exact.

For example, you can approximate it as

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

where  $h$  is small.

This is called a 'finite difference' approximation.

## Exercise 5

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 for the gradient and pass it to the function `newton_root` you coded earlier.

This is polymorphism: you now have two definition for the same function. They differ in the number of arguments.

## 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.

# 15. Lambda in object

A set of integers, with a test on which ones can be admitted:

```
// func/lambdafun.cpp
#include <functional>
using std::function;
    /* ... */
class SelectedInts {
private:
    vector<int> bag;
    function< bool(int) >
        selector;
public:
    SelectedInts
        ( function< bool(int) >
          f ) {
        selector = f; };
};
```

```
void add(int i) {
    if (selector(i))
        bag.push_back(i);
};

int size() {
    return bag.size(); };

std::string string() {
    std::string s;
    for ( int i : bag )
        s += to_string(i)+" ";
    return s;
};
```

## 16. Illustration

Code:

```
1 // func/lambdafun.cpp
2 cout << "Give a divisor: ";
3 cin >> divisor; cout << '\n';
4 cout << ".. using " << divisor
5     << '\n';
6 auto is_divisible =
7     [divisor] (int i) -> bool {
8     return i%divisor==0; };
9 SelectedInts multiples(
10     is_divisible );
11 for (int i=1; i<50; ++i)
12     multiples.add(i);
```

Output:

```
Give a divisor:
.. using 7
Multiples of 7:
7 14 21 28 35 42 49
```



## **Advanced topics**

# 17. Capture by value

Normal capture is by value:

Code:

```
1 // func/lambda capture.cpp
2 int one=1;
3 auto increment_by_n =
4     [one] ( int input ) -> int {
5     return input+one;
6 };
7 cout << increment_by_n (5) <<
8     '\n';
9 cout << increment_by_n (12) <<
10    '\n';
11 cout << increment_by_n (25) <<
12    '\n';
```

Output:

```
6
13
26
```

## 18. Capture by reference

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

```
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;
```

(See the algorithm header, section ??.)

# 19. Lambdas vs function pointers

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

Code:

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

Output:

```
C ptr: 6
Lambda: 6
```

## 20. Use in algorithms

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

See later.