3. C++ advanced (III): Functors and Lambda

What do we learn today?

- Functors: objects with overloaded function operator `()`.
- Closures
- Lambda-Expressions: syntactic sugar
- Captures
- Function type variables

Functors: Motivation

A simple output filter

```
template <typename T, typename Function>
void filter(const T& collection, Function f){
    for (const auto& x: collection)
        if (f(x)) std::cout << x << " ";
    std::cout << "\n";
}
```

(`filter` works if the first argument offers an iterator and if the second argument can be applied to elements with a result that can be converted to bool.)

Functors: Motivation

```
template <typename T, typename Function>
void filter(const T& collection, Function f);

template <typename T>
bool even(T x){
    return x % 2 == 0;
}

std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
filter(a,even<int>); // output: 2,4,6,16
```

Functor: Object with Overloaded Operator ()

```
class GreaterThan{
  int value; // state
  public:
  GreaterThan(int x):value{x}{}

  bool operator() (int par) const {
    return par > value;
  }
};
```

A Functor is a callable object. Can be understood as a stateful function.

```
std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
int value=8;
filter(a,GreaterThan(value)); // 9,11,16,19
```


Functor: object with overloaded operator ()

```
template <typename T>
class GreaterThan{
    T value;
public:
    GreaterThan(T x):value{x}{}

    bool operator() (T par) const{
        return par > value;
    }
};
```

(this also works with a template, of course)

```
std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
int value=8;
filter(a,GreaterThan<int>(value)); // 9,11,16,19
```


The same with a Lambda-Expression

```
std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
int value=8;

filter(a, [value](int x) {return x > value;} );
```


Sum of Elements – Old School

```
std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
int sum = 0;
for (auto x: a)
  sum += x;
std::cout << sum << std::endl; // 83
```

Sum of Elements – with Functor

```
template <typename T>
struct Sum{
    T value = 0;

    void operator() (T par){ value += par; }
};

std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
Sum<int> sum;
// for_each copies sum: we need to copy the result back
sum = std::for_each(a.begin(), a.end(), sum);
std::cout << sum.value << std::endl; // 83
```


Sum of Elements – with References[4]

```
template <typename T>
struct SumR{
    T& value;
    SumR (T& v):value{v} {}

    void operator() (T par){ value += par; }
};

std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
int s=0;
SumR<int> sum{s};
// cannot (and do not need to) assign to sum here
std::for_each(a.begin(), a.end(), sum);
std::cout << s << std::endl; // 83
```

[4] Of course this works, very similarly, using pointers

Sum of Elements – with Λ

```
std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};

int s=0;

std::for_each(a.begin(), a.end(), [&s] (int x) {s += x;} );

std::cout << s << std::endl;
```


Sorting by Different Order

```
// pre: i >= 0
// post: returns sum of digits of i
int q(int i){
    int res =0;
    for(;i>0;i/=10)
        res += i % 10;
    return res;
}

std::vector<int> v {10,12,9,7,28,22,14};
std::sort (v.begin(), v.end(),
  [] (int i, int j) { return q(i) < q(j);}
);
```

Now $v = 10, 12, 22, 14, 7, 9, 28$ (sorted by sum of digits)

Lambda-Expressions in Detail

```
[value] (int x) ->bool {return x > value;}
```

capture — parameters — return type — statement

Closure

```
[value] (int x) ->bool {return x > value;}
```

- Lambda expressions evaluate to a temporary object – a closure
- The closure retains the execution context of the function - the captured objects.
- Lambda expressions can be implemented as functors.

Simple Lambda Expression

```
[]()->void {std::cout << "Hello World";}
```

call:

```
[]()->void {std::cout << "Hello World";}();
```

assignment:

```
auto f = []()->void {std::cout << "Hello World";};
```


Minimal Lambda Expression

```
[]{}
```

- Return type can be inferred if no or only one return statement is present.[5]

  ```
  []() {std::cout << "Hello World";}
  ```

- If no parameters and no explicit return type, then () can be omitted.

  ```
  []{std::cout << "Hello World";}
  ```

- `[...]` can never be omitted.

[5] Since C++14 also several returns possible, provided that the same return type is deduced

Examples

```
[](int x, int y) {std::cout << x * y;} (4,5);
```

Output: 20

Examples

```
int k = 8;
auto f = [](int& v) {v += v;};
f(k);
std::cout << k;
```

Output: 16

Examples

```
int k = 8;
auto f = [](int v) {v += v;};
f(k);
std::cout << k;
```

Output: 8

Capture – Lambdas

For Lambda-expressions the capture list determines the context accessible

Syntax:

- `[x]`: Access a copy of x (read-only)
- `[&x]`: Capture x by reference
- `[&x,y]`: Capture x by reference and y by value
- `[&]`: Default capture all objects by reference in the scope of the lambda expression
- `[=]`: Default capture all objects by value in the context of the Lambda-Expression

Capture – Lambdas

```
int elements=0;
int sum=0;
std::for_each(v.begin(), v.end(),
  [&] (int k) {sum += k; elements++;} // capture all by reference
)
```


Capture – Lambdas

```
template <typename T>
void sequence(vector<int> & v, T done){
  int i=0;
  while (!done()) v.push_back(i++);
}

vector<int> s;
sequence(s, [&] {return s.size() >= 5;} )
```

now v = 0 1 2 3 4

The capture list refers to the context of the lambda expression.

Capture – Lambdas

When is the value captured?

```
int v = 42;
auto func = [=] {std::cout << v << "\n"};
v = 7;
func();
```

Output: 42

Values are assigned when the lambda-expression is created.

Capture – Lambdas

(Why) does this work?

```
class Limited{
  int limit = 10;
 public:
  // count entries smaller than limit
  int count(const std::vector<int>& a){
    int c = 0;
    std::for_each(a.begin(), a.end(),
        [=,&c] (int x) {if (x < limit) c++;}
    );
    return c;
  }
};
```

The `this` pointer is implicitly copied by value

Capture – Lambdas

```
struct mutant{
  int i = 0;
  void do(){ [=] {i=42;}();}
};

mutant m;
m.do();
std::cout << m.i;
```

Output: 42

The `this` *pointer* is implicitly copied by value

Lambda Expressions are Functors

```
[x, &y] () {y = x;}
```

can be implemented as

```
unnamed {x,y};
```

with

```
class unnamed {
  int x; int& y;
  unnamed (int x_, int& y_) : x (x_), y (y_) {}
  void operator () () {y = x;}
};
```


Lambda Expressions are Functors

```
[=] () {return x + y;}
```

can be implemented as

```
unnamed {x,y};
```

with

```
class unnamed {
  int x; int y;
  unnamed (int x_, int y_) : x (x_), y (y_) {}
  int operator () () const {return x + y;}
};
```


Polymorphic Function Wrapper `std::function`

```
#include <functional>

int k= 8;
std::function<int(int)> f;
f = [k](int i){ return i+k; };
std::cout << f(8); // 16
```

can be used in order to store lambda expressions.

Other Examples

`std::function<int(int,int)>`;

`std::function<void(double)>` ...

http://en.cppreference.com/w/cpp/utility/functional/function

Example

```
template <typename T>
auto toFunction(std::vector<T> v){
  return [v] (T x) -> double {
    int index = (int)(x+0.5);
    if (index < 0) index = 0;
    if (index >= v.size()) index = v.size()-1;
    return v[index];
  };
}
```

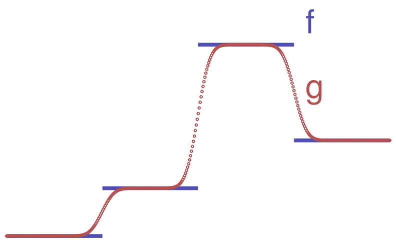

Example

```
auto Gaussian(double mu, double sigma){
    return [mu,sigma](double x) {
        const double a = ( x - mu ) / sigma;
        return std::exp( -0.5 * a * a );
    };
}

template <typename F, typename Kernel>
auto smooth(F f, Kernel kernel){
  return [kernel,f] (auto x) {
        // compute convolution ...
        // and return result
  };
}
```


Example

```
std::vector<double> v {1,2,5,3};
auto f = toFunction(v);
auto k = Gaussian(0,0.1);
auto g = smooth(f,k);
```



Conclusion

- Functors allow to write functional programs in C++. Lambdas are syntactic sugar to simplify this.
- With functors/lambdas classic patters from functional programming (e.g. map / filter /reduce) can be applied in C++.
- In combination with templates and the type inference (`auto`) very powerful functions can be stored in variables. Functions can even return functions (so called higher order functions).

Today's exercise

- Apply lambda expressions in a map/filter context on a vector of data.
- Implement a function (a lambda) that approximates the derivative of a given function.