

# 15. C++ advanced (III): Functors and Lambda

---

Generic Programming: higher order functions

# 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";
}
```

Question: when does a call of the `filter` template function work?

# 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";
}
```

Question: when does a call of the `filter` template function work?

Answer: works if the first argument offers an iterator and if the second argument can be applied to elements of the iterator with a return value 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
```

# Context and Overview

- Recent C++-lecture: make code parametric on the data it operates
  - `Pair<T>` for element types `T`
  - `print<C>` for iterable containers `C`
- Now: make code parameteric on a (part of) the algorithm
  - `filter(container, predicate)`
  - `apply(signal, transformation/filter)`
- We learn about
  - Higher-order functions: fucntions that take functions as arguments
  - Functors: objects with overloaded function operator `()`.
  - Lambda-Expressions: anonymous functors (syntactic sugar)
  - Closures: lambdas that capture their environment

# Functors: Motivation

```
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";
}
```

- Requirements on **f**: must be callable/applicable (...)
- **f** must be a kind of function  $\Rightarrow$  **filter** is a function that takes a function as argument
- A function taking (or returning) a function is called a **higher order function**
- Higher order functions are parameteric in their functionality (or they generate functions)

# What if...

the filter should be more flexible:

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

```
template <typename T>
bool largerThan(T x, T y){
    return x > y;
}
```

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

# What if...

the filter should be more flexible:

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

```
template <typename T>  
bool largerThan(T x, T y){  
    return x > y;  
}
```

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

```
int val = 8;
```

```
filter(a, largerThan<int>( ?, val)); (No, this does not exist)
```



# Functor: Object with Overloaded Operator ()

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

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

```
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
```

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

# 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
```

# Observations

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

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

Need to give the predicate a **name**

- Often unnecessary, many are used only once
- Descriptive names not always possible
- Distance (in the code) between declaration and use

**Overhead:** stateful predicates as functors

- cumbersome for what is ultimately only **par > value**

# The same with a Lambda-Expression

Anonymous functions with **lambda-expressions**:

```
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;} );
```

That is just **syntactic sugar**, from which the compiler generates a suitable functor.

# Interlude: Sorting with Custom Comparator

- `std::sort` is generic
  - in the iterator-type
  - in the values iterated over
  - in the used comparator

```
std::sort (v.begin(), v.end(), std::less());
```

- The comparator returns **true** if the elements are ordered as wished.

# 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);}
);
```

# 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  $v =$

# 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   statement  
                                  type

# 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";}
```

# Simple Lambda Expression

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

call:

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

# 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.<sup>18</sup>

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

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

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

- [...] can never be omitted.

---

<sup>18</sup>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:

# 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:

# 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:

# Examples

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

Output: 8

# Commonly Used: `std::foreach`

- Common task: iterate over a container and do something with each element

```
for (auto& name: names) {  
    std::cout << name << ' ' ;  
}
```

- This pattern is typically implemented as higher-order function `for_each`

```
for_each(names, [](auto name) {std::cout << name << " ";});
```

# 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 – Old School

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

- Task: increase **sum** for each call in **for\_each**
- Problem: **for\_each** requires access to the context (here **sum**)

Other example **filter**: store each element of a vector into a different vector according to some condition.

# 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 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
```

- Ok: solves the problem but does not operate on the **sum** variable

# Sum of Elements – with References

```
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
```

# Sum of Elements – with $\Lambda$

```
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;
```

# 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

```
std::vector<int> v = {1,2,4,8,16};

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

std::cout << "sum=" << sum << " elements=" << elements << std::endl;
```

Output:

# Capture – Lambdas

```
std::vector<int> v = {1,2,4,8,16};

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

std::cout << "sum=" << sum << " elements=" << elements << std::endl;
```

Output: sum=31 elements=5

# 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 =

# 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



# 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:

# 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;
    }
};
```

# 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 action(){ [=] {i=42;}();}  
};
```

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

Output:

# Capture – Lambdas

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

```
mutant m;  
m.action();  
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];
    };
}
```

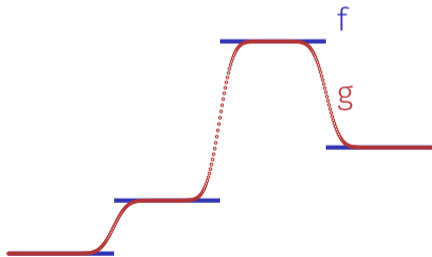
# Example

```
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

- Higher-order functions are parametric in their functionality: more flexible → more widely applicable → more code reuse
- Being able to pass around functions means being able to pass around entire computations.
- Lambda expressions facilitate the use of “one-off” functions, which are often used in combination with higher-order functions.
- Returning lambdas enables implementing function generators, that can generate whole families of functions.
- In C++, lambda expressions desugare into function objects (functors).
- Higher-order functions and lambdas are an important, and nowadays mainstream, building block of the functional programming paradigm.