CS 3460

Introduction to Lambdas

Three Functional C++ topics

- Functors
- Function Types
- Lambdas

Functors

- A *function object*, a *functor*
- A class that overloads the parenthesis `()` operator

Functors

- A *function object*, a *functor*
- A class that overloads the parenthesis `()` operator

```
#pragma once
#include <cstdint>
#include <vector>

class VectorSum
{
  public:
    std::uint64_t operator()(const std::vector<std::uint8_t>& data);
};
```

```
#include "VectorSum.hpp"

std::uint64_t VectorSum::operator()(const std::vector<std::uint8_t>& data)
{
    std::uint64_t total{ 0 };

    for (auto& value : data)
    {
        total += value;
    }

    return total;
}
```

Functors

- Let's see how to use the functor

```
#include "VectorSum.hpp"

#include <iostream>
#include <vector>

int main()
{
    std::vector<std::uint8_t> primes{2, 3, 5, 7};
    std::vector<std::uint8_t> evens{ 2, 4, 6, 8, 10 };
    VectorSum sum;

    std::cout << "The sum of the primes is " << sum(primes) << std::endl;
    std::cout << "The sum of the evens is " << sum(evens) << std::endl;

    return 0;
}
```

- Looks just like we are calling a function!

Functors and The Standard Library

- Functors look cool, but so what?
- They allow algorithm parameterization
- The STL uses functors for what it calls a *predicate*
  - A predicate returns a `bool` result
- Three kinds of predicates in the standard library
  - ***Generator*** : functor with no arguments
  - ***Unary function*** : functor with one argument
  - ***Binary function*** : functor with two arguments

Functors and The Standard Library

- The STL has an algorithm `std::all_of`
  - Accepts a sequence of values & unary predicate
  - Tests if all values result in a `true` result from the predicate
    - If all `true`, algorithm returns `true`, `false` otherwise

Functors and The Standard Library

- The STL has an algorithm `std::all_of`
  - Accepts a sequence of values & unary predicate
  - Tests if all values result in a `true` result from the predicate
    - If all `true`, algorithm returns `true`, `false` otherwise
- Let's give it a try...

Standard Library – `std::all_of`

- Let's give it a try…

```
class ValidateEven
{
  public:
    bool operator()(std::uint8_t value) { return value % 2 == 0; }
};

ValidateEven validate;
std::cout << "The primes are all even: " <<
     std::all_of(primes.begin(), primes.end(), validate) << std::endl;
std::cout << "The evens are all even: " <<
     std::all_of(evens.begin(), evens.end(), validate) << std::endl;
```

- This is functional polymorphism
- ***Note:*** Functors are no longer necessary with lambdas in C++ 11; we are discussing them as the lead up to lambdas

Function Types

- From it’s origin in C, have always had ability to define pointers to functions, yes, pointers to functions!
- Syntax for a function pointer looks messy at first glance
- We aren’t going to use this, because of C++ 11

Function Types

- C++ has the notion of a *callable*
  - Anything that can be given to `std::invoke` as a parameter
  - Basically anything that looks and acts like a function is a callable; functions, functors, and lambdas
- Often need to define a type for a callable
  - Could use function pointers
  - Better is to use `std::function`
    - `<functional>`

Function Types – Example

- Let's write our own "all of" function

```
bool myAllOf(const std::vector<std::uint8_t>& data, std::function<bool(std::uint8_t)> test)
{
    bool allTrue{ true };
    for (auto value : data)
    {
        if (test(value) == false)
        {
            allTrue = false;
            break;
        }
    }

    return allTrue;
}
```

- Note the second parameter
  - A function that returns a `bool`
  - Accepts a `std::uint8_t`

Function Types – Example

- Let's use it with our `ValidateEven` functor from before...

```
ValidateEven validate;
std::cout << "The primes are all even: " << myAllOf(primes, validate) << std::endl;
std::cout << "The evens are all even: " << myAllOf(evens, validate) << std::endl;
```

Function Types – Example

- Let's use it with our `ValidateEven` functor from before...

```
ValidateEven validate;
std::cout << "The primes are all even: " << myAllOf(primes, validate) << std::endl;
std::cout << "The evens are all even: " << myAllOf(evens, validate) << std::endl;
```

- Can also use a regular function...

```
bool validateEven(std::uint8_t value) { return value % 2 == 0; }

std::cout << "The primes are all even: " << myAllOf(primes, validateEven) << std::endl;
std::cout << "The evens are all even: " << myAllOf(evens, validateEven) << std::endl;
```

Lambdas

- Showed up in C++ 11; enhancements since
- Terms *lambda*, *lambda function*, *lambda expression* all used interchangeably
- What is a lambda?
  - Ability to define a function object type with no name
- Let’s start with an example, next slide...

Lambdas

- Using `std::function` to represent the type

```
std::function<void(void)> myLambda = []() { std::cout << "My first lambda!" << std::endl; };
```

- `std::function` is not the actual type, it is a compatible type
  - Type of a lambda is *ineffible*

Lambdas

- Using `std::function` to represent the type

```
std::function<void(void)> myLambda = []() { std::cout << "My first lambda!" << std::endl; };
```

- `std::function` is not the actual type, it is a compatible type
  - Type of a lambda is *ineffible*
- Using `auto` to deduce the type

```
auto myLambda = []() { std::cout << "My first lambda!" << std::endl; };
```

Lambdas – Syntax Detail

```
[capture](parameters) mutable-specification exception-specification -> return type { body }
```

- **capture** : declare which data is captured and how
- **parameters** : same as function parameters
- **mutable specification** : enable call to non-const members
- **exception specification** : indicate if no exceptions occur
- **return type** : same as functions; generally inferred
- **body** : statements to be executed

Lambda – Another Example

- Let's use a lambda to validate the numbers in a vector are even, using our own "all of" function.

```
std::cout << "The primes are all even: " <<
     myAllOf(primes, [](std::uint8_t value) { return value % 2 == 0; })
     << std::endl;
```

Lambda Capture

- External state can be captured at the time a lambda is instantiated; this is called a *closure*
- A closure can live beyond the scope in which it was defined
- Therefore, need a way to remember state from the original scope for later use
- This is the purpose of the *capture clause* of a lambda

Lambda Capture

- `[]` : Nothing captured
- `[=]` : Capture all used variables by value (copy)
- `[&]` : Capture all used variables by reference (alias)
- `[data]` : Capture `data` by value
- `[&data]` : Capture `data` by reference
- `[=, &data]` : Capture all by value, except `data` by reference
- `[&, data]` : Capture all by reference, except `data` by value

Easy Lambda Mistake

- Lambda closure can outlive the scope in which it was created; caution is necessary
- Capture a local variable by reference, then use that variable after scope is gone...bad things happen!

```
std::function<std::uint32_t()> makeLambda(std::uint32_t value)
{
    std::uint32_t local{ value };
    return [&]() { return local; };
}

auto badLambda = makeLambda(8);
auto result = badLambda();
std::cout << result << std::endl;
```

- Can solve with capture by-value `[=]` instead of `[&]`

Generic Lambdas

- Where one or more parameters are templated using the `auto` keyword

```
auto isGreater = [](auto a, auto b) { return a > b; };

std::cout << "is 10 greater than 20? "
     << (isGreater(10, 20) ? "yes " : "no") << std::endl;

std::cout << "is 1.1234 greater than 1? "
     << (isGreater(1.1234, 1) ? "yes " : "no") << std::endl;

std::cout << "is 'Logan' better than 'Amalga'? "
     << (isGreater("Logan", "Amalga") ? "yes" : "no") << std::endl;
```

- This may be thought of as being similar to

```
class // anonymous
{
public:
     template<typename T, typename R>
     bool operator()(T a, R b) { return a > b; }
} isGreater;
```