Introduction to Functional C++

Introduction

C++ is a multi-paradigm language. One of the paradigms expressed in the language is a functional approach to solving problems. This section presents a progression of topics that lead into how lambdas are utilized. The topic begins with *functors*, followed by *function types*, and finishes with *lambdas*.

Functors

A *function object* is a class that overloads the parenthesis `()` operator, also called a *functor*. Overloading the parenthesis operator enables an instance of the class to look like it is being called as a function. Lets start with a simple example.

--- VectorSum.hpp ---

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

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

--- VectorSum.cpp ---

```
#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;
}
```

This class overloads the `()` operator. The `()` operator may be overloaded with zero to any number (and type) of arguments. There is no general semantic meaning for what this operator should do, unlike the mathematical and relational operators. Instead, any meaning that is contextually appropriate may be used. For `VectorSum`, the meaning is to compute and return the sum of all the elements in an `std::vector`.

--- main.cpp ---

```
#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;
}
```

This code declares an instance of a `VectorSum` object, which is then used in the console output statements. The `()` operator is used to invoke the summing code. Remember, an overloaded operator ***is*** a function. Rather than invoking it as `sum(primes)`, it can be invoked as `sum.operator()(primes)`. The `sum(primes)` and `sum(evens)` parts of the code are calls to functions.

`VectorSum` is a functor.

Functors and The Standard Library

We know what a functor is, so what, what is their usefulness?

Functors are used throughout the standard library to provide functional parameterization of various algorithms. The standard library uses functors as what it calls a *predicate*. A predicate is a functor that returns a `bool` result. There are three kinds of predicates in the standard library.

1. **Generator** – A functor that has no arguments.
2. **Unary function** – A functor that has one argument.
3. **Binary function** – A functor that has two arguments.

The standard library has an algorithm called `std::all_of` which takes a sequence of values, along with a unary function predicate. The algorithm tests each value in the sequence with the predicate and if all values in the sequence return `true`, based on the predicate, `std::all_of` returns `true`, otherwise it returns `false`.

The next code sample shows the code for a `ValidateEven` functor that returns `true` if a value is even. The functor is then used in `std::all_of` to verify all the values in an `std::vector` are even.

```
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 known as functional polymorphism. The `std::all_of` algorithm is polymorphic, what it does depends on the functor given to it.

The need for functors has declined (been eliminated?) with the advent of lambdas (presentation coming soon) in C++ 11. I want to discuss function types next, then dive into lambdas and see how they basically eliminate the need for functors. I present functors because it provides a better background and more detailed look (and I think better understanding), of lambdas.

Function Types

C++ has always had the ability to create and manage pointers to functions (yes, pointers to functions!), because of its history in the C language. In other words, it is possible to define a pointer type that points to a function. Using the address-of operator `&`, the memory address of a function can be obtained and assigned to the function pointer. Then, the function pointer can be used to invoke the function. The syntax for this looks a bit messy at first glance, but isn't that terrible once understood. However, I'm not going to cover it in these notes, because it is unnecessary with C++. Because of function pointers, C++ has always had functional polymorphism, as just demonstrated with functors.

C++ has the notion of a *callable*. Specifically, since C++ 17, the definition of a callable is anything that can be given to `std::invoke` as a parameter. Basically, anything that looks and acts like a function is a callable. A function is a callable, a functor is a callable, and as we'll see soon, a lambda is a callable.

Sometimes it is necessary to define a type for a callable. Perhaps we want to write some function polymorphic code, parameterized by passing in a function as a constructor parameters. We could define a function pointer type, as suggested above, but there is a "cleaner" syntax, the use of `std::function`. The standard library `<functional>` provides `std::function`.

An example best serves to demonstrate how to use `std::function`. We'll write our own "all of" function, that works similarly to `std::all_of`.

```
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;
}
```

The second parameter is the one of interest. `std::function` is templated, with the type being the signature (plus return value) of the function. The way to read the declaration is, "`test` is a function that returns a `bool` and accepts an `std::uint8_t` as a parameter". The template type for `std::function` can be any valid function signature, along with the return type (including void). This function can be used with the same `ValidateEven` functor, as shown next.

```
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;
```

The type given to `std::function` does not have to be a function, just a callable. A regular function may also be used, such as:

```
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

Lambda functions or expressions are one of the most interesting, and welcome, additions to the C++ language, showing up with C++ 11, with additional enhancements since. The terms *lambda function* and *lambda expression* are used interchangeably throughout these notes; both also simplified to *lambda*. A lambda is the ability to define a function object that has no name, a so-called anonymous function object.

The best place to start is with a simple lambda function.

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

… alternatively…

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

The right-hand side of the statement (right of the =) is the lambda definition. At first glance, the syntax looks quite unusual. The lambda is assigned to an `std::function` variable. Rather than using the `std::function` type, it is probably preferred to use `auto`, as the type of a lambda is created on-the-fly by the compiler, not exactly an `std::function`, but a compatible callable that can be stored in an `std::function` variable.

Lambdas in Java require a *functional interface* (an interface with one method). This is because Java has a very strong type system, where it requires all types not defined as part of the language to be defined in code. This is different from C++, which is considered a strongly typed language, where it will create a new type on-the-fly for a lambda. The type of a C++ lambda is not known, it is *ineffible* (too great to express), but may be assigned to a compatible `std::function` type.

The lambda is invoked just like a function:

```
myLambda();
```

Syntax

The basic lambda syntax is:

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

There are six sections that compose a lambda function: capture, parameters, mutable specification, exception specification, return type, and the body. Of these, only the *capture* and *body* are required, all others are optional. Each of these is briefly described next.

- **capture** – Used to indicate which variables declared outside the scope of the lambda are accessible within the lambda. This is more fully explored in the next section entitled “Lambda Capture”.
- **parameters** – Lambdas are used in the context of some code calling them, passing parameters into the lambda, just as like passing parameters into a function. This section identifies the parameter types and names.
- **mutable specification** – The `mutable` keyword can be used here to inform the lambda to call non-const member functions of captured (by copy) parameters.
- **exception specification** – A `throw()` or `noexcept` keyword can be used here to indicate the lambda does not throw an exception. If this specification is used and the code within the lambda has code that generates an exception, a compiler warning results.
- **return type** – The return type for the lambda, in the same sense that a function returns a value of some type. Generally this is not necessary as the compiler is able to infer the return type. If there is more than one return statement within the lambda, the return type becomes necessary.
- **body** – This is where the statements that perform the desired function are placed.

With this knowledge of lambdas, rather than writing a functor to validate the numbers in a vector are even, a lambda can be used.

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

This code shows a lambda can be defined and used as the predicate (a unary function in this case) parameter. Functionally, no pun intended, there isn't a difference between a function and lambda, just that the lambda syntax is more convenient.

Lambda Capture

A lambda is an object that captures external state at the time it is instantiated (when it forms a *closure*). Because a lambda instantiation (I will simply say lambda from here on out to also mean lambda instantiation or closure) may be passed in as a parameter or live beyond the scope in which it was defined, there needs to be a way to remember data from the original scope for later use. That is the purpose of the capture clause of a lambda. Consider each of the following capture examples:

`[]` Nothing is captured.

`[=]` Capture all used variables by value (copy).

`[&]` Capture all used variables by reference (alias).

`[data]` Capture the variable data by value.

`[&data]` Capture the variable data by reference.

`[=, &data]` Capture all used variables by value, except for data, which is captured by reference.

`[&, data]` Capture all used variables by reference, except for data, which is captured by value.

Because a lambda is an object with a lifetime that may exist longer than the scope in which it was defined, some caution is necessary in its use. For example, a lambda may be returned from a function, with the lambda having captured a local function variable by reference. Later on, when the lambda is invoked and it attempts to access the reference capture, it will either use an invalid value or cause an exception because the scope for the referenced local variable no longer exists. The next code sample demonstrates this.

```
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;
```

The lambda function captures the local variable `local` by reference, with the lambda then returned from the function. When this code is executed some number gets displayed, but probably not `8`. On my computer `3435978836` is displayed. The program did not throw an exception, it returned a value stored at a no longer valid memory location. Because it didn't thrown an exception, you can imagine the potential for a subtle bug to be introduced. If the capture has been by value, `return [=]() { return local; };`, the problem would not have happened.

Generic Lambdas

A generic lambda is one in which one or more of the parameters are templated using the `auto` keyword. Consider the following code:

```
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;
```

You can think of the generic lambda creating a class that looks like:

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

The reason for two template parameters is because `a` and `b` might be different types, but must have the `>` operator defined for them.

It is not required for all parameters to a generic lambda are generic.

The following line of code causes a compiler error because no > operator is defined for an `int` and `std::string`.

```
std::cout << (isGreater(1, "this") ? "yes" : "no") << std::endl;
```

It's that simple!