“Functional-Style” Programming and Functional Objects in C++

Presented by Dr. Ofri Sadowsky, CoreCpp Meetup, 27/6/2019

Ofri Sadowsky is an employee of Harman

This lecture

Is about…

- Motivation for using “functional style” programming in C++
- Explaining some of the tools that C++ offers for functional-style programming
- Focus mostly on “functional” objects
- Suggesting some practical tips about the use and misuse of functional-style programming in C++

Is NOT about…

- A scientific explanation of functional programming
- A complete usage guide of functionals in C++
- Absolute rules

Part 1: Functional Programming in a Sunflower-Seed Shell*

* Less than a nutshell

What is functional programming?

From Wikipedia:

- “functional programming is a programming paradigm… that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.”
- “a function's return value depends only on its arguments, so calling a function with the same value for an argument always produces the same result. This is in contrast to imperative programming where, in addition to a function's arguments, global program state can affect a function's resulting value.”
- “Programming in a functional style can be accomplished in languages that are not specifically designed for functional programming, such as with Perl, PHP, C++11, and Kotlin.”

What is functional programming?

From Wikipedia:

- “functional programming is a programming paradigm… that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.”
- “a function's return value depends only on its arguments, so calling a function with the same value for an argument always produces the same result. This is in contrast to imperative programming where, in addition to a function's arguments, global program state can affect a function's resulting value.”
- “Programming in a functional style can be accomplished in languages that are not specifically designed for functional programming, such as with Perl, PHP, C++11, and Kotlin.”

What is functional programming?

From Wikipedia:

- “functional programming is a programming paradigm… that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.”
- “a function's return value depends only on its arguments, so calling a function with the same value for an argument always produces the same result. This is in contrast to imperative programming where, in addition to a function's arguments, global program state can affect a function's resulting value.”
- “Programming in a functional style can be accomplished in languages that are not specifically designed for functional programming, such as with Perl, PHP, C++11, and Kotlin.”

What is functional programming?

From Wikipedia:

- “functional programming is a programming paradigm… that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.”
- “a function's return value depends only on its arguments, so calling a function with the same value for an argument always produces the same result. This is in contrast to imperative programming where, in addition to a function's arguments, global program state can affect a function's resulting value.”
- “Programming *in a functional style* can be accomplished in languages that are not specifically designed for functional programming, such as with Perl, PHP, *C++11*, and Kotlin.”

Recursion in Functional Programming

- Functional Programming has no state variables, and no assignment operations.
- Therefore, no iterators.
- This means heavy reliance on recursion.
- Few people use it in daily practice, but it’s useful as an abstraction and as an alternative thought direction.
- In some cases, like metaprogramming tasks, it’s the only working solution.

Case Study 1: Vector Operations

```
double dotProduct(Vector const & v1,
    Vector const & v2, size_t size)
{
  if (size == 0) {
    return 0.0;
  }
  else {
    return dotProduct(v1,v2,size-1)+
      v1[size-1] * v2[size-1];
  }
}
```

```
class Vector;
/// A hypothetical class that
/// represents an algebraic
/// vector of doubles
```

```
bool areEqual (Vector const & v1,
    Vector const & v2, size_t size)
{
  if (size == 0) {
    return true;
  }
  else {
    return areEqual(v1,v2,size-1) &&
      (v1[size-1] == v2[size-1]);
  }
}
```

Now, don’t these look sort of similar?

Vector Operations in Template Form

```
template<class TOut, class TIn>
using BinaryFunc = TOut (*) (TIn const &, TIn const &);

template<class TRes, class TInter>
TRes engine(BinaryFunc<TRes, TInter> reducer,
   BinaryFunc<TInter, double> oper, TRes const & emptyRes,
   Vector const & v1, Vector const & v2, size_t size)
{
   if (size == 0) {
     return emptyRes;

   }
   else {
     return reducer(
        engine(reducer, oper, emptyRes, v1, v2, size-1),
        oper(v1[size-1], v2[size-1]));
   }
}
```

Concretizing Vector Operations

```
double mult(double d1, double d2)
{ return d1 * d2; }

double add(double d1, double d2)
{ return d1 + d2; }

double dotProduct(Vector const & v1,
    Vector const & v2, size_t size)
{
  return engine(add, mult, 0.0,
    v1, v2, size);
}
```

```
bool eq(double d1, double d2)
{ return (d1 == d2); }

bool and(bool b1, bool b2)
{ return (b1 && b2); }

bool areEqual (Vector const & v1,
    Vector const & v2, size_t size)
{
  return engine(and, eq, true,
    v1, v2, size);
}
```

Case Study 2: Numerical Integration

```
using functional = double (*)(double);

double integrate(functional f, double begin, double end,
                 double step) {
  double result = 0.0;
  for (size_t i = 0; begin + double(i) * step < end; i++) {
    result += f(begin + double(i) * step) * step;
  }
  return result;
}

double square(double x) { return x * x; }

double integral = integrate(&square, 0.0, 3.0, 0.001);
```

Nice! But can we integrate an integrator?

The answer is yet to come.

Functional Objects

- In a simple description, functional objects are *objects* that behave like *functions*.
  - A user can "call" on the object (invoke), passing arguments, and receive a return value.
  - In C++, this is achieved by overloading `operator()` for a class.
- In a broader sense, one can argue that with *any* method (or function), if one of the parameters can be "invoked", that parameter is a "functional object".
  - Consider the Template Method design pattern (coming soon).
  - The difference between the TM Pattern and operator() is only syntactical.

Functional Objects

- Unlike functions in FP, functional objects have a state (i.e. member variables) that can affect the outcome of invocation.
  - As long as the object is constant, the function outcome for the same input stays the same.
  - If the state of the object changes between invocations, it may produce a different outcome (hidden function arguments)
  - The invocation can have side effects that change the state of the functional object (write new values to members) or of other objects that it interacts with.
- Functional objects are not part of functional programming in the classical definition.
  - Is it good or bad?

The Template Method Pattern (side note)

```
class BaseAlgorithm {
public:
  double run() {
    double value = getSpecialData();
    return value * value;
  }

protected:
  virtual double getSpecialData() = 0;
};
```

Case Study 2: Integrator Functional Object

```
using functional =
  double (*)(double);

double integrate(functional f,
    double begin, double end,
    double step);
```

```
class Integrator {
public:
  Integrator(functional f,
    double begin, double step);

  double operator()(double x) const
  {
    return integrate(mIntegrand,
      mBegin, x, mStep);
  }

private:
  functional mIntegrand;
  double     mBegin;
  double     mStep;
};
```

Case Study 2: Integrator Functional Object with Template

```
using functional =
  double (*)(double);

double integrate(functional f,
  double b, double e, double s);

class Integrator {
public:
  Integrator(functional f,
    double b, double s);

  double operator()(double x) const;

private:
  // see above...
};
```

```
template<class TIntegrand>
double templateIntegrate(
  TIntegrand const & integrand,
  double b, double e, double s)
{
  double result = 0.0;
  for (size_t i = 0;
    b + double(i) * s < e; i++) {
      result += integrand(
          b + double(i) * s) * s;
  }
  return result;
}
```

What Did We Learn So Far?

- In classical functional programming, *everything* is a function.
- The simplest form of a functional object in C++ (and C!) is a function pointer.
- In C++ the notion of a functional object can be expressed by an overloaded operator() or, in a broader sense, by overridden virtual methods.
- Functional objects are an essential element of generic programming, e.g.
  - Code template (functions, classes)
  - Design patterns (Template Method, Observer, …)

Part 2: C++11 Functional Objects

Functional Object Categories in C++11+

Category	Form / Example
Global function pointers	`using functional = double (*)(double);`

Member function pointers	`class MyClass { double someMethod(double); };`
	`using MFunc = double (MyClass::*)(double);`
	`MFunc f = &MyClass::someMethod;`
	`MyClass obj;`
	`double v = (obj.*f)(5.0);`

"Crafted" functional class	`class Integrator { double operator()(double); };`
Virtual methods	`class Algorithm { virtual double f(double) = 0; };`

std::bind objects	`auto integrator = std::bind(integrate, square,`
	`         0.0, std::placeholders::_1, 0.001);`
	`double integral = integrator(3.0);`

Lambda objects	`auto f = [](double x) { return x * x; };` `double v = f(5.0);`

std::function	`using functor = std::function<double(double)>;`
	`functor f = /* most of the above...*/;`
	`double v = f(5.0);`

Argument Binding

- “Bind” is an “operator” on a functional object and other parameters, which returns another functional object:

```
double add(double x, double y) { return x + y; }

auto add10 = std::bind(add, _1, 10.0); // _1 is a placeholder for an argument passed to add10
```

- In this example, “add” is the *bound* functional object, and “add10” is the *binding* functional object.
- `add10.operator()` takes one parameter and forwards it to the bound functional along with a bound argument, which happens to be 10.0. ➔ Effectively, “add10” is a unary functional object.
- The roots of “bind” go back to Lambda Calculus – a theoretical model of computability and functional programming.
- The C++ syntax and usage rules of `std::bind` are (subjectively) cryptic and often confusing.
- Clang-Tidy recommends to “prefer a lambda to std::bind”, and I join.

Lambda Objects

- The term “Lambda” comes from Lambda Calculus (LC), mentioned above, where it represents a functional.
- C++11+ defines a new syntax (“syntactic sugar”) for instantiation of functional objects, which can replace most of the hand-crafted overloads of operator(), and simultaneously add *bind* capabilities. These objects are “lambda objects” or just “lambdas”.
- C++ lambdas are slightly abusing the original LC lambdas because they are real objects, they can mutate a global program state, and can even have a mutable state of their own.
- But it’s a catchy name and the abuse is small, and if they’re immutable, well, it’s close enough.

What’s in a Lambda?

- Much more detail and examples in Andreas Fertig’s presentation of Core C++ 2019: https://www.andreasfertig.info/talks_dl/afertig-corecpp-2019-cpp-lambdas-demystified.pdf
- Here’s the short of it.

```
double add(double x, double y) { return x + y; }

void myFunction() {
  double num = 10.0;

  auto innerLambda = [num](double y)
  {
    return add(num, y);
  }

  double sum = innerLambda(5.0);
  std::cout << “sum = “ << sum << std::endl;
}
```

capture: `[num]`

parameters: `(double y)`

body: `{ return add(num, y); }`

invocation: `innerLambda(5.0)`

What’s in a Lambda?

- Much more detail and examples in Andreas Fertig’s presentation of Core C++ 2019: https://www.andreasfertig.info/talks_dl/afertig-corecpp-2019-cpp-lambdas-demystified.pdf
- Here’s the short of it.

```
double add(double x, double y) { return x + y; }

void myFunction() {
  double num = 10.0;

  auto innerLambda = [num](double y)
  {
    return add(num, y);
  }

  double sum = innerLambda(5.0);
  std::cout << "sum = " << sum << std::endl;
}
```

capture

parameters

body

invocation

- *capture* defines simultaneously class members and class constructor.
- *parameters* define the signature of a public operator().
- *body* is the function body of operator()
- *auto* is required because the class name is compiler-generated and we cannot know it.
- *invocation* calls the operator() method for the lambda object.

Things to Remember about C++ Lambda

- Lambdas define classes and their instantiations.
- After the instantiation, the lambda is a full-blown object.
- The captured entities (if they exist) are members of the lambda object.
- A lambda object can be copied (including copy of the captures), moved (including move of the captures), or passed by reference.
- The concrete type of the lambda is inaccessible, so if it is passed to a generic algorithm (like integrate), the type must be abstracted.
  - Write the algorithm as a template, or
  - Wrap the lambda object by a std::function object.

std::function

```
template<class R, class... Args>
class function<R(Args...)>;
```

```
using functor = std::function<double(double)>;
functor f = /* most of the above */;
double v = f(5.0);
```

- A specialization of std::function is a well-defined and accessible type.
- Any instance of this type can host (or contain) any *callable object* that matches the type's signature.
  - Yes, different instances of the same std::function type can host callable objects of different types.
  - Yes, this can be source for much trouble…
- A std::function object is invokable, with the signature of operator() determined from the template specialization.

std::function – a Peek Under the Hood

What does it take to construct a std::function instance?

`template<class F> function(F f);` with F being an invokable type.

1. Allocate as much memory as needed to host an instance of F.
2. Construct an instance of F (copy or move from f).
3. Move the instance to the allocated memory (in-place move construct).
4. Keep pointers to lifetime-control member functions of F:
   - Copy constructor, in case one wishes to copy the hosting std::function instance (what about move?)
   - Destructor for the time of destructing the hosting std::function instance
   - operator() which will be called from the hosting instance's operator()
   - …

std::function – Observations

- Heavy-size object, expensive to construct, expensive to copy.
- Relatively cheap to invoke – use a member function pointer bound to an internally-stored instance.
- The content and the actual function code cannot be predicted before construction or deduced after the construction (type-erased).
- Some things are impossible.

```
class A {
public:
  void operator()() { callImpl(); }
  virtual void callImpl();
  double member1;
};
```

```
class B : public A {
public:
  void callImpl() override;
  char member2[100];
};
```

```
void foo(A const & obj) {
  std::function<void(void)> functor = obj;
  functor();
}
```

C++11 Functional Objects – Summary

- C++11 defines several types and syntaxes to simplify the definition and construction of functional objects:
  - std::bind
  - Lambdas
  - std::function
- All these types are full-blown objects (lambdas can be simpler).
- They loosely represents functionals in the FP paradigm, but with some important differences (can affect global and own state, std::function can be reassigned).
- Using them has benefits and prices

Part 3: A Few Handy Rules

A Few Handy Rules

1. Prefer a lambda to std::bind.
   - Recommended by Clang-Tidy, already mentioned.
2. Prefer a lambda to hand-crafted classes with overloaded operator()
   - Lambdas simplify your life and prevent funny corner cases.
3. In my “generic” algorithm, choose lambda or std::function?
   - Lambda usually requires the algorithm to be templated over the concrete type of the functional. It’s often more efficient but exposes the intrinsics.
   - std::function is type-erased and supports better abstraction and encapsulation, with some cost of performance.

A Few Handy Rules

4. Refrain from nested lambdas (lambda defined inside another lambda)
   - Significant obfuscation
   - Usually can be refactored into methods or separate nested object captured into the larger lambda.
5. Capture with consideration, try to be minimalistic (no [&] [=])
   - Reduce lambda size and avoid funny side effects
6. Reduce copy/pass-by-value of callable objects
   - Possibly high performance price

A Few Handy Rules

7. Choose wisely between functional objects and plain-ol' polymorphism.
   - Many times, if you know exactly how a function should behave, polymorphism is the right answer for you.
   - If you have a collection of functional objects with equal captures or common content, a real class is probably a better answer.