Functional-style programming

HCMC C++ users meetup

Germán Diago Gómez

April 24th, 2016

Goals of this talk

- Introduce some functional-style patterns in C++.

Goals of this talk

- Introduce some functional-style patterns in C++.
- Show some examples combined with the STL.

Goals of this talk

- Introduce some functional-style patterns in C++.
- Show some examples combined with the STL.
- Present some more advanced examples of its use at the end.

Non-goals

- Not a pure-functional Haskell-style programming talk.

Main traits

- Use of immutable data.

Main traits

- Use of immutable data.
- Use of pure functions.

Main traits

- Use of immutable data.
- Use of pure functions.
- Use of lazy evaluation.

Main traits

- Use of immutable data.
- Use of pure functions.
- Use of lazy evaluation.
- Heavy use of recursivity.

Main traits

- Use of immutable data.
- Use of pure functions.
- Use of lazy evaluation.
- Heavy use of recursivity.
- Functions as data. They can be parameters to other functions.

Main traits

- Use of immutable data.
- Use of pure functions.
- Use of lazy evaluation.
- Heavy use of recursivity.
- Functions as data. They can be parameters to other functions.
- Functions can also be returned.

Main traits

- Use of immutable data.
- Use of pure functions.
- Use of lazy evaluation.
- Heavy use of recursivity.
- Functions as data. They can be parameters to other functions.
- Functions can also be returned.
- Composability.

Three important functional algorithms

- `map` → `std::transform` in STL.

Three important functional algorithms

- `map` → `std::transform` in STL.
- `filter` → `std::remove_if` in STL.

Three important functional algorithms

- `map` → `std::transform` in STL.
- `filter` → `std::remove_if` in STL.
- `reduce` → `std::accumulate` in STL.

Three important functional algorithms

- `map` → `std::transform` in STL.
- `filter` → `std::remove_if` in STL.
- `reduce` → `std::accumulate` in STL.
- They are the base of many powerful patterns and algorithms.

Why functional programming

- Multithreaded code becomes much easier to deal with (no locks needed).

Why functional programming

- Multithreaded code becomes much easier to deal with (no locks needed).
- Code much easier to parallelize automatically.

Why functional programming

- Multithreaded code becomes much easier to deal with (no locks needed).
- Code much easier to parallelize automatically.
- Higher-order functions enable algorithms customization.

Why functional programming

- Multithreaded code becomes much easier to deal with (no locks needed).
- Code much easier to parallelize automatically.
- Higher-order functions enable algorithms customization.
  - Without rewriting algorithms for special cases.

Why functional programming

- Multithreaded code becomes much easier to deal with (no locks needed).
- Code much easier to parallelize automatically.
- Higher-order functions enable algorithms customization.
    - Without rewriting algorithms for special cases.
- Higher order functions enable other useful patterns.

Why functional programming

- Multithreaded code becomes much easier to deal with (no locks needed).
- Code much easier to parallelize automatically.
- Higher-order functions enable algorithms customization.
  - Without rewriting algorithms for special cases.
- Higher order functions enable other useful patterns.
- Pure functions: can be memoized.

Main traits

- Use of function objects.

Main traits

- Use of function objects.
- Use of lambdas.

Main traits

- Use of function objects.
- Use of lambdas.
- Creating callables that return other callables.

Main traits

- Use of function objects.
- Use of lambdas.
- Creating callables that return other callables.
- Pass function objects/lambdas as parameters, usually to STL algorithms.

What is a function object?

- A `struct` or `class`.

What is a function object?

- A `struct` or `class`.
- Implements the call operator `operator()`.

What is a function object?

- A `struct` or `class`.
- Implements the call operator `operator()`.
- Objects whose class/struct implements `operator()` can be called the same way as functions are called.

Why function objects are important

- The STL makes heavy use of them.

Why function objects are important

- The STL makes heavy use of them.
- Can carry state, unlike classic C style functions.

Why function objects are important

- The STL makes heavy use of them.
- Can carry state, unlike classic C style functions.
- Efficient: Easier to inline than function pointers and pointers to members.

Why function objects are important

- The STL makes heavy use of them.
- Can carry state, unlike classic C style functions.
- Efficient: Easier to inline than function pointers and pointers to members.
    - Better code generation.

Why function objects are important

- The STL makes heavy use of them.
- Can carry state, unlike classic C style functions.
- Efficient: Easier to inline than function pointers and pointers to members.
  - Better code generation.
- If you understand function objects you understand lambdas.

Predicates

A predicate is a callable that returns true or false given some input parameter(s).

Example (Unary predicate function object)

```
struct is_negative {
  bool operator()(int n) const {
    return n < 0;
  }
};

std::cout << is_negative{}(-5);
```

Example (Unary predicate function object)

```
struct is_negative {
  bool operator()(int n) const {
    return n < 0;
  }
};

std::cout << is_negative{}(-5);
```

Output

```
1
```

Example (Binary predicate function object)

```
struct food {
  std::string food_name;
  double average_user_score;
};

struct more_delicious {
  bool operator()(food const & f1, food const & f2) const {
    return f1.average_user_score > f2.average_user_score;
  }
};

food const pho{"pho", 8.1}, com_tam{"com tam", 7.6};

std::cout << "Pho more declicious? -> "
<< more_delicious{}(pho, com_tam);
```

Example (Binary predicate function object)

```
struct food {
  std::string food_name;
  double average_user_score;
};

struct more_delicious {
  bool operator()(food const & f1, food const & f2) const {
    return f1.average_user_score > f2.average_user_score;
  }
};

food const pho{"pho", 8.1}, com_tam{"com tam", 7.6};

std::cout << "Pho more declicious? -> "
<< more_delicious{}(pho, com_tam);
```

Output

```
Pho more declicious? -> 1
```

Other function objects

- Ternary predicates could also exist.

Other function objects

- Ternary predicates could also exist.
  - The STL only uses unary and binary.

Other function objects

- Ternary predicates could also exist.
    - The STL only uses unary and binary.
- Not all function objects are necessarily predicates.

Other function objects

- Ternary predicates could also exist.
    - The STL only uses unary and binary.
- Not all function objects are necessarily predicates.
- Although in the STL predicates are very common in algorithms.

(Live demo)

Problems with function objects

- Verbose: must create a class always.

Problems with function objects

- Verbose: must create a class always.
- Usually used once and thrown away.

Problems with function objects

- Verbose: must create a class always.
- Usually used once and thrown away.
    - Write a class for one use only?

Alternatives to handcrafted fuction objects

- Use predefined function objects. STL: `std::less`, `std::multiplies` and many others. Insufficient.

Alternatives to handcrafted fuction objects

- Use predefined function objects. STL: `std::less`, `std::multiplies` and many others. Insufficient.
- Compose objects via `std::bind`. Composing with `std::bind` is complicated, less efficient than lambdas and potentially surprising. Avoid.

Alternatives to handcrafted fuction objects

- Use predefined function objects. STL: `std::less`, `std::multiplies` and many others. Insufficient.
- Compose objects via `std::bind`. Composing with `std::bind` is complicated, less efficient than lambdas and potentially surprising. Avoid.
- Make use of lambda functions.

Alternatives to handcrafted fuction objects

- Use predefined function objects. STL: `std::less`, `std::multiplies` and many others. Insufficient.
- Compose objects via `std::bind`. Composing with `std::bind` is complicated, less efficient than lambdas and potentially surprising. Avoid.
- Make use of lambda functions.
- We will focus on lambda functions.

Lambda functions

- Lambda functions are syntactic sugar for function objects.

Lambda functions

- Lambda functions are syntactic sugar for function objects.
- They are equally efficient.

Lambda functions

- Lambda functions are syntactic sugar for function objects.
- They are equally efficient.
- Not verbose.

Lambda functions

- Lambda functions are syntactic sugar for function objects.
- They are equally efficient.
- Not verbose.
- Two kinds in C++

Lambda functions

- Lambda functions are syntactic sugar for function objects.
- They are equally efficient.
- Not verbose.
- Two kinds in C++
  - Monomorphic lambdas: non-templated `operator()`.

Lambda functions

- Lambda functions are syntactic sugar for function objects.
- They are equally efficient.
- Not verbose.
- Two kinds in C++
  - Monomorphic lambdas: non-templated `operator()`.
  - Polymorphic lambdas: templated `operator()`.

Anatomy of a lambda function

- `[capture-list-opt](params) mutable-opt noexcept-opt -> ret_type { body }`.

Anatomy of a lambda function

- `[capture-list-opt](params) mutable-opt noexcept-opt -> ret_type { body }`.
- The `[]` is called the *lambda introducer*.

Anatomy of a lambda function

- `[capture-list-opt](params) mutable-opt noexcept-opt -> ret_type { body }`.
- The `[]` is called the *lambda introducer*.
- The *capture list* is optional.

Anatomy of a lambda function

- `[capture-list-opt](params) mutable-opt noexcept-opt -> ret_type { body }`.
- The `[]` is called the *lambda introducer*.
- The *capture list* is optional.
- The *ret_type* is also optional, otherwise it is deduced from the body.

Anatomy of a lambda function

- `[capture-list-opt](params) mutable-opt noexcept-opt -> ret_type { body }`.
- The `[]` is called the *lambda introducer*.
- The *capture list* is optional.
- The *ret_type* is also optional, otherwise it is deduced from the body.
- Lambdas generate by default *lambda_class::operator() const*.

Given the following code...

```
std::vector<int> data = {1, 3, 5, 2, 1, 28};

int threshold = 20;
std::partition(std::begin(data), std::end(data), 0,
               [threshold](int a)
               { return a < threshold; });
```

- Every lambda function generates a different compiler struct type.

. . . the compiler generates something like this

```
struct __anon_object {
    __anon_object(int _threshold) : //capture variables
        threshold(_threshold) {}

    decltype(auto) operator(int a, int b) const {
        return a < threshold; }

    int const threshold; //captured by value
};

std::vector<int> data = {1, 3, 5, 2, 1, 28};

int threshold = 20;
std::partition(std::begin(data), std::end(data), 0,
               __anon_object{threshold});
```

- Every lambda generated is unique even if they contain the same code, captures, etc.

Predicates with lambdas

(Demo)

Lambdas are very powerful

- Can capture the environment (stateful).

Lambdas are very powerful

- Can capture the environment (stateful).
- They are not verbose as function objects.

Lambdas are very powerful

- Can capture the environment (stateful).
- They are not verbose as function objects.
- Lambdas can save a lot of code but still keep it efficient.

Problem: partial function application

We have a function that renders some text into a target screen with a given size and orientation.

```
void render_text(Screen & target, int font_size,
                 int pos_x, int pos_y,
                 Orientation text_orientation,
                 std::string const & text);
```

We want to call this function all the time with the same parameters to render different text, but it becomes very tedious: many parameters must be passed.

Solution

```
Screen screen; //Non-copyable

auto render_at_top_left = [=, &screen](std::string const & text) {
    return render_text(screen,
                       80,
                       k_left_side_screen,
                       k_top_screen,
                       Orientation::Horizontal,
                       text);
};

render_at_top_left("Hello, world!");
```

Problem: timing functions

We want to measure the time it takes to run a function or piece of code and some functions from some APIs.

- We do not have access to the source code of the functions we want to measure.

```
using namespace std;

  template <class Func>
  auto timed_func(Func && f) {
     return [f = forward<Func>(f)](auto &&... args) {
       auto init_time = sc::high_resolution_clock::now();
       f(forward<decltype(args)>(args)...);
       auto total_exe_time = sc::high_resolution_clock::now()
       - init_time;

       return sc::duration_cast<sc::milliseconds>
       (total_exe_time).count();
     };
  }

  int main() {
    vector<int> vec; vec.reserve(2'000'000);
    int num = 0;
    while (cin >> num) vec.push_back(num);
    auto timed_sort = timed_func([&vec]() { sort(begin(vec),
    end(vec)); });
```

```
  cout << "Sorting 2,000,000 numbers took "
  << timed_sort() << " milliseconds.\n";
}
```

Problem: map/reduce data.

Calculate the average of the squares of some series of data

Solution

```
using namespace std;

vector<int> vec(20);
iota(begin(vec), end(vec), 1);
vector <int> res(20);

transform(std::begin(vec), end(vec), std::begin(res),
[](int val) { return val * val; });
auto total = accumulate(begin(res), end(res), 0,
[](int acc, int val) { return val + acc; });

std::cout << total << '\n';
```

Holding any callable

- Lambdas and function objects have their own concrete type.

Holding any callable

- Lambdas and function objects have their own concrete type.
- No common class even if can be invoked with same parameters.

Holding any callable

- Lambdas and function objects have their own concrete type.
- No common class even if can be invoked with same parameters.
- This is good because it enables inlining easily...

Holding any callable

- Lambdas and function objects have their own concrete type.
- No common class even if can be invoked with same parameters.
- This is good because it enables inlining easily...
- ...but bad because you cannot store collections of callables or do indirect calls to them.

Holding any callable

- Lambdas and function objects have their own concrete type.
- No common class even if can be invoked with same parameters.
- This is good because it enables inlining easily...
- ...but bad because you cannot store collections of callables or do indirect calls to them.
- C++ has function objects, lambdas, pointers to members, function pointers...

Holding any callable

- Lambdas and function objects have their own concrete type.
- No common class even if can be invoked with same parameters.
- This is good because it enables inlining easily...
- ...but bad because you cannot store collections of callables or do indirect calls to them.
- C++ has function objects, lambdas, pointers to members, function pointers...
    - With different call syntaxes.

Holding any callable

- Lambdas and function objects have their own concrete type.
- No common class even if can be invoked with same parameters.
- This is good because it enables inlining easily...
- ...but bad because you cannot store collections of callables or do indirect calls to them.
- C++ has function objects, lambdas, pointers to members, function pointers...
  - With different call syntaxes.
- How can we store arbitrary callables in containers?

std::function<FuncSignature>

- `std::function` can store arbitrary callables.

std::function<FuncSignature>

- `std::function` can store arbitrary callables.
- the callables that can store depend on the signature given in its template parameter.

std::function<FuncSignature>

- `std::function` can store arbitrary callables.
- the callables that can store depend on the signature given in its template parameter.
- can capture anything callable: functions, member functions, function objects, lambdas...

std::function use cases

- Use when you do not know what you will store until run-time.

std::function use cases

- Use when you do not know what you will store until run-time.
- Use to store callables in containers. Command pattern is implemented by std::function directly.

std::function use cases

- Use when you do not know what you will store until run-time.
- Use to store callables in containers. Command pattern is implemented by std::function directly.
- Prefer `auto` to `std::function` when possible for your variables, though. More efficient.

```
struct Calculator {
  int current_result = 5;
  int add_with_context(int a, int b) {
    return a + b + current_result;
  }
};

int add(int a, int b) { return a + b; }

int main() {
  std::function<int (int, int)> bin_op;
  bin_op = add; //Store plain function
  std::cout << bin_op(3, 5) << std::endl;
  Calculator c;
  bin_op = std::multiplies<int>{}; //Store function object
  std::cout << bin_op(3, 5) << std::endl;
  //Call member function capturing calculator object:
  bin_op = [&c](int a, int b) { return c.add_with_context(a, b); };
  std::cout << bin_op(3, 5) << std::endl;
}
```

Thank you